

NATIONAL COMPUTER SECURITY CENTER

AD-A277 642



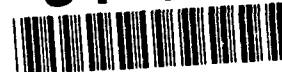
A Guide to Understanding
Security Testing
and
Test Documentation

in Trusted Systems



July 1993

94-09740



DTIC QUALITY 3 1 1994

Approved for Public Release:
Distribution Unlimited

94 3 31 039

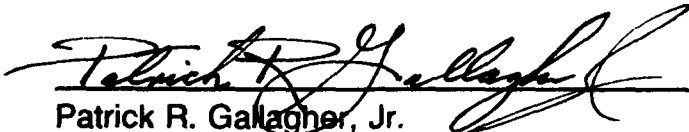
FOREWORD

The National Computer Security Center is issuing ***A Guide to Understanding Security Testing and Test Documentation in Trusted Systems*** as part of the "Rainbow Series" of documents our Technical Guidelines Program produces. In the Rainbow Series, we discuss in detail the features of the Department of Defense Trusted Computer System Evaluation Criteria (DoD 5200.28-STD) and provide guidance for meeting each requirement. The National Computer Security Center, through its Trusted Product Evaluation Program, evaluates the security features of commercially produced computer systems. Together, these programs ensure that users are capable of protecting their important data with trusted computer systems.

The specific guidelines in this document provide a set of good practices related to security testing and the development of test documentation. This technical guideline has been written to help the vendor and evaluator community understand what deliverables are required for test documentation, as well as the level of detail required of security testing at all classes in the Trusted Computer System Evaluation Criteria.

As the Director, National Computer Security Center, I invite your suggestions for revision to this technical guideline. We plan to review this document as the need arises.

National Computer Security Center
Attention: Chief, Standard, Criteria and Guidelines Division
9800 Savage Road
Fort George G. Meade, MD 20755-6000



Patrick R. Gallagher, Jr.

Director

National Computer Security Center

January, 1994

ACKNOWLEDGMENTS

Special recognition and acknowledgment for his contributions to this document are extended to Virgil D. Gligor, University of Maryland, as primary author of this document.

Special thanks are extended to those who enthusiastically gave of their time and technical expertise in reviewing this guideline and providing valuable comments and suggestions. The assistance of C. Sekar Chandersekaran, IBM and Charles Bonneau, Honeywell Federal Systems, in the preparation of the examples presented in this guideline is gratefully acknowledged.

Special recognition is extended to MAJ James P. Gordon, U.S. Army, and Leon Neufeld as National Computer Security Center project managers for this guideline.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

FOREWORD	i
ACKNOWLEDGMENTS	iii
1. INTRODUCTION	1
1.1 PURPOSE	1
1.2 SCOPE	1
1.3 CONTROL OBJECTIVES	3
2. SECURITY TESTING OVERVIEW	5
2.1 OBJECTIVES	5
2.2 PURPOSE	5
2.3 PROCESS	5
2.3.1 System Analysis	5
2.3.2 Functional Testing	6
2.3.3 Security Testing	7
2.4 SUPPORTING DOCUMENTATION	8
2.5 TEST TEAM COMPOSITION	9
2.6 TEST SITE	10
3. SECURITY TESTING - APPROACHES, DOCUMENTATION, AND EXAMPLES	11
3.1 TESTING PHILOSOPHY	11
3.2 TEST AUTOMATION	13
3.3 TESTING APPROACHES	15
3.3.1 Monolithic (Black-Box) Testing	16
3.3.2 Functional-Synthesis (White- Box) Testing	18
3.3.3 Gray-Box Testing	21
3.4 RELATIONSHIP WITH THE TCSEC SECURITY TESTING REQUIREMENTS	25
3.5 SECURITY TEST DOCUMENTATION	30
3.5.1 Overview	30
3.5.2 Test Plan	30
3.5.2.1 Test Conditions	31
3.5.2.2 Test Data	34
3.5.2.3 Coverage Analysis	36
3.5.3 Test Procedures	37

Table of Contents

3.5.4 Test Programs	38
3.5.5 Test Log	39
3.5.6 Test Report	39
3.6 SECURITY TESTING OF PROCESSORS' HARDWARE/FIRMWARE PROTECTION MECHANISMS	39
3.6.1 The Need for Hardware/Firmware Security Testing	40
3.6.2 Explicit TCSEC Requirements for Hardware Security Testing ..	41
3.6.3 Hardware Security Testing vs. System Integrity Testing	43
3.6.4 Goals, Philosophy, and Approaches to Hardware Security Testing	43
3.6.5 Test Conditions, Data, and Coverage Analysis for Hardware Security Testing	44
3.6.5.1 Test Conditions for Isolation and Noncircumventability Testing	45
3.6.5.2 Text Conditions for Policy-Relevant Processor Instructions	46
3.6.5.3 Tests Conditions for Generic Security Flaws	46
3.6.6 Relationship between Hardware/Firmware Security Testing and the TCSEC Requirements	48
3.7 TEST PLAN EXAMPLES	50
3.7.1 Example of a Test Plan for "Access"	53
3.7.1.1 Test Conditions for Mandatory Access Control of "Access"	53
3.7.1.2 Test Data for MAC Tests	54
3.7.1.3 Coverage Analysis	55
3.7.2 Example of a Test Plan for "Open"	59
3.7.2.1 Test Conditions for "Open"	60
3.7.2.2 Test Data for the AccessGraph Dependency Condition ..	60
3.7.2.3 Coverage Analysis	62
3.7.3 Examples of a Test Plan for "Read"	63
3.7.3.1 Test Conditions for "Read"	64
3.7.3.2 Test Data for the Access-Check Dependency Condition	64
3.7.3.3 Coverage Analysis	68
3.7.4 Examples of Kernel Isolation Test Plans	69
3.7.4.1 Test Conditions	69
3.7.4.2 Test Data	70
3.7.4.3 Coverage Analysis	71
3.7.5 Examples of Reduction of Cyclic Test Dependencies	72
3.7.6 Example of Test Plans for Hardware/Firmware Security Testing	75

3.7.6.1 Test Conditions for the Ring Crossing Mechanism	76
3.7.6.2 Test Data	76
3.7.6.3 Coverage Analysis	79
3.7.7 Relationship with the <i>TCSEC</i> Requirements	80
4. COVERT CHANNEL TESTING	87
4.1 COVERT CHANNEL TEST PLANS	87
4.2 AN EXAMPLE OF A COVERT CHANNEL TEST PLAN	89
4.2.1 Test Plan for the Upgraded Directory Channel	90
4.2.1.1 Test Condition	90
4.2.1.2 Test Data	90
4.2.1.3 Coverage Analysis	92
4.2.2 Test Programs	92
4.2.3 Test Results	92
4.3 RELATIONSHIP WITH THE <i>TCSEC</i> REQUIREMENTS	93
5. DOCUMENTATION OF SPECIFICATION-TO-CODE CORRESPONDENCE	95
APPENDIX	97
1 Specification-to-Code Correspondence	97
2 Informal Methods for Specification-to-Code Correspondence	98
3 An Example of Specification-to-Code Correspondence	102
GLOSSARY	111
REFERENCES	119

1. INTRODUCTION

The National Computer Security Center (NCSC) encourages the widespread availability of trusted computer systems. In support of this goal the *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)* was created as a metric against which computer systems could be evaluated. The NCSC published the *TCSEC* on 15 August 1983 as CSC-STD-001-83. In December 1985, the Department of Defense (DoD) adopted it, with a few changes, as a DoD Standard, DoD 5200.28-STD. [13] DoD Directive 5200.28, "Security Requirements for Automatic Data Processing (ADP) Systems," requires that the *TCSEC* be used throughout the DoD. The NCSC uses the *TCSEC* as a standard for evaluating the effectiveness of security controls built into ADP systems. The *TCSEC* is divided into four divisions: D, C, B, and A. These divisions are ordered in a hierarchical manner with the highest division (A) being reserved for systems providing the best available level of assurance. Within divisions C and B there are a number of subdivisions known as classes. In turn, these classes are also ordered in a hierarchical manner to represent different levels of security.

1.1 PURPOSE

Security testing is a requirement for *TCSEC* classes C1 through A1. This testing determines that security features for a system are implemented as designed and that they are adequate for the specified level of trust. The *TCSEC* also requires test documentation to support the security testing of the security features of a system. The *TCSEC* evaluation process includes security testing and evaluation of test documentation of a system by an NCSC evaluation team. *A Guide to Understanding Security Testing and Test Documentation for Trusted Systems* will assist the operating system developers and vendors in the development of computer security testing and testing procedures. This guideline gives system developers and vendors suggestions and recommendations on how to develop testing and testing documentation that will be found acceptable by an NCSC Evaluation Team.

1.2 SCOPE

TCSEC classes C1 through A1 assurance is gained through security testing and the accompanying test documentation of the ADP system. Security testing and test documentation ensures that the security features of the system are implemented as

designed and are adequate for an application environment. This guideline discusses the development of security testing and test documentation for system developers and vendors to prepare them for the evaluation process by the NCSC. This guideline addresses, in detail, various test methods and their applicability to security and accountability policy testing. The Trusted Computing Base (TCB) isolation, noncircumventability testing, processor testing, and covert channel testing methods are examples.

This document provides an in-depth guide to security testing. This includes the definitions, writing and documentation of the test plans for security and a brief discussion of the mapping between the formal top-level specification (FTLS) of a TCB and the TCB implementation specifications. This document also provides a standard format for test plans and test result presentation. Extensive documentation of security testing and specification-to-code correspondence arise both during a system evaluation and, more significantly, during a system life cycle. This guideline addresses evaluation testing, not life-cycle testing. This document complements the security testing guideline that appears in Section 10 of the *TCSEC*.

The scope and approach of this document is to assist the vendor in security testing and in particular functional testing. The vendor is responsible for functional testing, not penetration testing. If necessary, penetration testing is conducted by an NCSC evaluation team. The team collectively identifies penetration vulnerabilities of a system and rates them relative to ease of attack and difficulty of developing a hierarchy penetration scenario. Penetration testing is then conducted according to this hierarchy, with the most critical and easily executed attacks attempted first [17].

This guideline emphasizes the testing of systems to meet the requirements of the *TCSEC*. *A Guide to Understanding Security Testing and Test Documentation for Trusted Systems* does not address the testing of networks, subsystems, or new versions of evaluated computer system products. It only addresses the requirements of the *TCSEC*.

Information in this guideline derived from the requirements of the *TCSEC* is prefaced by the word "shall." Recommendations that are derived from commonly accepted good practices are prefaced by the word "should." The guidance contained herein is intended to be used when conducting and documenting security functional testing of an operating system. The recommendations in this document

are not to be construed as supplementary requirements to the *TCSEC*. The *TCSEC* is the only metric against which systems are to be evaluated.

Throughout this guideline there are examples, illustrations, or citations of test plan formats that have been used in commercial product development. The use of these examples, illustrations, and citations is not meant to imply that they contain the only acceptable test plan formats. The selection of these examples is based solely on their availability in computer security literature. Examples in this document are not to be construed as the only implementations that will satisfy the *TCSEC* requirements. The examples are suggestions of appropriate implementations.

1.3 CONTROL OBJECTIVES

The *TCSEC* and DoD 5200.28-M [14] provide the control objectives for security testing and documentation. Specifically these documents state the following.

"Component's Designated Approving Authorities, or their designees for this purpose . . . will assure: . . .

"4. Maintenance of documentation on operating systems (O/S) and all modifications thereto, and its retention for a sufficient period of time to enable tracing of security-related defects to their point of origin or inclusion in the system.

"5. Supervision, monitoring, and testing, as appropriate, of changes in an approved ADP System that could affect the security features of the system, so that a secure system is maintained. . .

"6. Proper disposition and correction of security deficiencies in all approved ADP Systems, and the effective use and disposition of system housekeeping or audit records, records of security violations or security-related system malfunctions, and records of tests of the security features of an ADP System.

"7. Conduct of competent system Security Testing and Evaluation (ST&E), timely review of system ST&E reports, and correction of deficiencies needed to support conditional or final approval or disapproval of an ADP system for the processing of classified information.

"8. Establishment, where appropriate, of a central ST&E coordination point for the maintenance of records of selected techniques, procedures, standards, and tests used in testing and evaluation of security features of ADP systems which may be suitable for validation and use by other Department of Defense components."

Section 5 of the *TCSEC* gives the following as the Assurance Control Objective:

"The third basic control objective is concerned with guaranteeing or providing confidence that the security policy has been implemented correctly and that the protection critical elements of the system do, indeed, accurately mediate and enforce the intent of that policy. By extension, assurance must include a guarantee that the trusted portion of the system works only as intended. To accomplish these objectives, two types of assurance are needed. They are life-cycle assurance and operational assurance.

"Life-cycle assurance refers to steps taken by an organization to ensure that the system is designed, developed, and maintained using formalized and rigorous controls and standards. Computer systems that process and store sensitive or classified information depend on the hardware and software to protect that information. It follows that the hardware and software themselves must be protected against unauthorized changes that could cause protection mechanisms to malfunction or be bypassed completely. For this reason, trusted computer systems must be carefully evaluated and tested during the design and development phases and reevaluated whenever changes are made that could affect the integrity of the protection mechanisms. Only in this way can confidence be provided that the hardware and software interpretation of the security policy is maintained accurately and without distortion." [13]

2. SECURITY TESTING OVERVIEW

This section provides the objectives, purpose, and a brief overview of vendor and NCSC security testing. Test team composition, test site location, testing process, and system documentation are also discussed.

2.1 OBJECTIVES

The objectives of security testing are to uncover all design and implementation flaws that enable a user external to the TCB to violate security and accountability policy, isolation, and noncircumventability.

2.2 PURPOSE

Security testing involves determining (1) a system security mechanism adequacy for completeness and correctness and (2) the degree of consistency between system documentation and actual implementation. This is accomplished through a variety of assurance methods such as analysis of system design documentation, inspection of test documentation, and independent execution of functional testing and penetration testing.

2.3 PROCESS

A qualified NCSC team of experts is responsible for independently evaluating commercial products to determine if they satisfy TCSEC requirements. The NCSC is also responsible for maintaining a listing of evaluated products on the NCSC Evaluated Products List (EPL). To accomplish this mission, the NCSC Trusted Product Evaluation Program has been established to assist vendors in developing, testing, and evaluating trusted products for the EPL. Security testing is an integral part of the evaluation process as described in the *Trusted Product Evaluations—A Guide For Vendors*. [18]

2.3.1 System Analysis

System analysis is used by the NCSC evaluation team to obtain a complete and in-depth understanding of the security mechanisms and operations of a vendor's product prior to conducting security testing. A vendor makes available to an NCSC team any information and training to support the NCSC team members in their understanding of the system to be tested. The NCSC team will become intimately

familiar with a vendor's system under evaluation and will analyze the product design and implementation, relative to the *TCSEC*.

System candidates for *TCSEC* ratings B2 through A1 are subject to verification and covert channel analyses. Evaluation of these systems begins with the selection of a test configuration, evaluation of vendor security testing documentation, and preparation of an NCSC functional test plan.

2.3.2 Functional Testing

Initial functional testing is conducted by the vendor and results are presented to the NCSC team. The vendor should conduct extensive functional testing of its product during development, field testing, or both. Vendor testing should be conducted by procedures defined in a test plan. Significant events during testing should be placed in a test log. As testing proceeds sequentially through each test case, the vendor team should identify flaws and deficiencies that will need to be corrected. When a hardware or software change is made, the test procedure that uncovered the problem should then be repeated to validate that the problem has been corrected. Care should be taken to verify that the change does not affect any previously tested procedure. These procedures also should be repeated when there is concern that flaws or deficiencies exist. When the vendor team has corrected all functional problems and the team has analyzed and retested all corrections, a test report should be written and made a part of the report for review by the NCSC test team prior to NCSC security testing.

The NCSC team is responsible for testing vendor test plans and reviewing vendor test documentation. The NCSC team will review the vendor's functional test plan to ensure it sufficiently covers each identified security mechanism and explanation in sufficient depth to provide reasonable assurance that the security features are implemented as designed and are adequate for an application environment. The NCSC team conducts its own functional testing and, if appropriate, penetration testing after a vendor's functional testing has been completed.

A vendor's product must be free of design and implementation changes, and the documentation to support security testing must be completed before NCSC team functional testing. Functional security testing is conducted on C1 through A1 class

systems and penetration testing on B2, B3, and A1 class systems. The NCSC team may choose to repeat any of the functional tests performed by the vendor and/or execute its own functional test. During testing by the NCSC team, the team informs the vendor of any test problems and provides the vendor with an opportunity to correct implementation flaws. If the system satisfies the functional test requirements, B2 and above candidates undergo penetration testing. During penetration testing the NCSC team collectively identifies penetration vulnerabilities in the system and rates them relative to ease of attack and difficulty in developing a penetration hierarchy. Penetration testing is then conducted according to this hierarchy with the most critical and most easily executed attacks attempted first [17]. The vendor is given limited opportunity to correct any problems identified [17]. When opportunity to correct implementation flaws has been provided and corrections have been retested, the NCSC team documents the test results. The test results are input which support a final rating, the publication of the Final Report and the EPL entry.

2.3.3 Security Testing

Security testing is primarily the responsibility of the NCSC evaluation team. It is important to note, however, that vendors shall perform security testing on a product to be evaluated using NCSC test methods and procedures. The reason for vendor security testing is two-fold: First, any TCB changes required as a result of design analysis or formal evaluation by the NCSC team will require that the vendor (and subsequently the evaluation team) retest the TCB to ensure that its security properties are unaffected and the required changes fixed the test problems. Second, any new system release that affects the TCB must undergo either a re-evaluation by the NCSC or a rating-maintenance evaluation by the vendor itself. If a rating maintenance is required, which is expected to be the case for the preponderant number of TCB changes, the security testing responsibility, including all the documentation evidence, becomes a vendor's responsibility—not just that of the NCSC evaluation team.

Furthermore, it is important to note that the system configuration provided to the evaluation team for security testing should be the same as that used by the vendor itself. This ensures that consistent test results are obtained. It also allows the evaluation team to examine the vendor test suite and to focus on areas deemed to be insufficiently tested. Identifying these areas will help speed the security testing of

a product significantly. (An important implication of reusing the vendor's test suite is that security testing should yield repeatable results.)

When the evaluation team completes the security testing, the test results are shown to the vendor. If any TCB changes are required, the vendor shall correct or remove those flaws before TCB retesting by the NCSC team is performed.

2.4 SUPPORTING DOCUMENTATION

Vendor system documentation requirements will vary, and depending on the TCSEC class a candidate system will be evaluated for, it can consist of the following:

Security Features User's Guide. It describes the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another. This may be used to identify the protection mechanisms that need to be covered by test procedures and test cases.

Trusted Facility Manual. It describes the operation and administration of security features of the system and presents cautions about functions and privileges that should be controlled when running a secure facility. This may identify additional functions that need to be tested.

Design Documentation. It describes the philosophy of protection, TCB interfaces, security policy model, system architecture, TCB protection mechanisms, top level specifications, verification plan, hardware and software architecture, system configuration and administration, system programming guidelines, system library routines, programming languages, and other topics.

Covert Channel Analysis Documentation. It describes the determination and maximum bandwidth of each identified channel.

System Integrity Documentation. It describes the hardware and software features used to validate periodically the correct operation of the on-site hardware and firmware elements of the TCB.

Trusted Recovery Documentation. It describes procedures and mechanisms assuring that after an ADP system failure or other discontinuity, recovery is obtained without a protection compromise. Information describing

procedures and mechanisms may also be found in the Trusted Facility Manual.

Test Documentation. It describes the test plan, test logs, test reports, test procedures, and test results and shows how the security mechanisms were functionally tested, covert channel bandwidth, and mapping between the FTLS and the TCB source code. Test documentation is used to document plans, tests, and results in support of validating and verifying the security testing effort.

2.5 TEST TEAM COMPOSITION

A vendor test team should be formed to conduct security testing. It is desirable for a vendor to provide as many members from its security testing team as possible to support the NCSC during its security testing. The reason for this is to maintain continuity and to minimize the need for retraining throughout the evaluation process. The size, education, and skills of the test team will vary depending on the size of the system and the class for which it is being evaluated. (See Chapter 10 of the TCSEC, "A Guideline on Security Testing.")

A vendor security testing team should be comprised of a team leader and two or more additional members depending on the evaluated class. In selecting personnel for the test team, it is important to assign individuals who have the ability to understand the hardware and software architecture of the system, as well as an appropriate level of experience in system testing. Engineers and scientists with backgrounds in electrical engineering, computer science and software engineering are ideal candidates for functional security testing. Prior experience with penetration techniques is important for penetration testing. A mathematics or logic background can be valuable in formal specifications involved in A1 system evaluation.

The NCSC test team is formed using the guidance of Chapter 10, in the TCSEC, "A Guideline on Security Testing." This chapter specifies test team composition, qualifications and parameters. Vendors may find these requirements useful recommendations for their teams.

2.6 TEST SITE

The location of a test site is a vendor responsibility. The vendor is to provide the test site. The evaluator's functional test site may be located at the same site at which the vendor conducted his functional testing. Proper hardware and software must be available for testing the configuration as well as appropriate documentation, personnel, and other resources which have a significant impact on the location of the test site.

3. SECURITY TESTING—APPROACHES, DOCUMENTATION, AND EXAMPLES

3.1 TESTING PHILOSOPHY

Operating systems that support multiple users require security mechanisms and policies that guard against unauthorized disclosure and modification of critical user data. The TCB is the principal operating system component that implements security mechanisms and policies that must itself be protected [13]. TCB protection is provided by a reference monitor mechanism whose data structures and code are isolated, noncircumventable, and small enough to be verifiable. The reference monitor ensures that the entire TCB is isolated and noncircumventable.

Although TCBs for different operating systems may contain different data structures and programs, they all share the isolation, noncircumventability, and verifiability properties that distinguish them from the rest of the operating system components. These properties imply that the security functional testing of an operating system TCB may require different methods from those commonly used in software testing for all security classes of the TCSEC.

Security testing should be done for TCBs that are configured and installed in a specific system and operate in a normal mode (as opposed to maintenance or test mode). Tests should be done using user-level programs that cannot read or write internal TCB data structures or programs. New data structures and programs should also not be added to a TCB for security testing purposes, and special TCB entry points that are unavailable to user programs should not be used. If a TCB is tested in the maintenance mode using programs that cannot be run at the user level, the security tests would be meaningless because assurance cannot be gained that the TCB performs user-level access control correctly. If user-level test programs could read, write or add internal TCB data structures and programs, as would be required by traditional instrumentation testing techniques, the TCB would lose its isolation properties. If user-level test programs could use special TCB entry points not normally available to users, the TCB would become circumventable in the normal mode of operation.

Security testing of operating system TCBs in the normal mode of operation using user-level test programs (which do not rely on breaching isolation and noncircumventability) should address the following problems of TCB verifiability

through security testing: (1) Coverage Analysis, (2) Reduction of Cyclic Test Dependencies, (3) Test Environment Independence, and (4) Repeatability of Security Testing.

(1) Coverage Analysis. Security testing requires that precise, extensive test coverage be obtained during TCB testing. Test coverage analysis should be based on coverage of test conditions derived from the Descriptive Top-Level Specification (DTLS)/Formal Top-Level Specification (FTLS), the security and accountability model conditions, the TCB isolation and noncircumventability properties, and the individual TCB-primitive implementation. Without covering such test conditions, it would be impossible to claim reasonably that the tests cover specific security checks in a demonstrable way. Whenever both DTLS and FTLS and security and accountability models are unavailable or are not required, test conditions should be derived from documented protection philosophy and resource isolation requirements [13]. It would be impossible to reasonably claim that the implementation of a specific security check in a TCB primitive is correct without individual TCB-primitive coverage. In these checks a TCB primitive may deal differently with different parameters. In normal-mode testing, however, using user-level programs makes it difficult to guarantee significant coverage of TCB-primitive implementation while eliminating redundant tests that appear when multiple TCB primitives share the same security checks (a common occurrence in TCB kernels).

The role of coverage analysis in the generation of test plans is discussed in Section 3.5.2, and illustrated in Sections 3.7.1.3–3.7.3.3.

(2) Reduction of Cyclic Test Dependencies. Comprehensive security testing suggests that cyclic test dependencies be reduced to a minimum or eliminated whenever possible. A cyclic test dependency exists between a test program for TCB primitive A and TCB primitive B if the test program for TCB primitive A invokes TCB primitive B, and the test program for TCB primitive B invokes TCB primitive A. The existence of cyclic test dependencies casts doubts on the level of assurance obtained by TCB tests. Cyclic test dependencies cause circular arguments and assumptions about test coverage and, consequently, the interpretation of the test results may be flawed. For example, the test program for TCB primitive A, which depends on the correct behavior of TCB primitive B, may not discover flaws in TCB primitive A because such flaws may be masked by the behavior of B, and vice versa. Thus, both the assumptions (1) that the TCB primitive B works correctly,

which must be made in the test program for TCB primitive A, and (2) that TCB primitive A works correctly, which must be made in the test program for TCB primitive B, are incorrect. The elimination of cyclic test dependencies could be obtained only if the TCB is instrumented with additional code and data structures—an impossibility if TCB isolation and noncircumventability are to be maintained in normal mode of operation.

An example of cyclic test dependencies, and of their removal, is provided in Section 3.7.5.

(3) Test Environment Independence. To minimize test program and test environment dependencies the following should be reinitialized for different TCB-primitive tests: user accounts, user groups, test objects, access privileges, and user security levels. Test environment initialization may require that the number of different test objects to be created and logins to be executed become very large. Therefore, in practice, complete TCB testing cannot be carried out manually. Testing should be automated whenever possible. Security test automation is discussed in Section 3.2.

(4) Repeatability of Security Testing. TCB verifiability through security testing requires that the results of each TCB-primitive test be repeatable. Without test repeatability it would be impossible to evaluate developers' TCB test suites independently of the TCB developers. Independent TCB testing may yield different outcomes from those expected if testing is not repeatable. Test repeatability by evaluation teams requires that test plans and procedures be documented in an accurate manner.

3.2 TEST AUTOMATION

The automation of the test procedures is one of the most important practical objectives of security testing. This objective is important for at least three reasons. First, the procedures for test environment initialization include a large number of repetitive steps that do not require operator intervention, and therefore, the manual performance of these steps may introduce avoidable errors in the test procedures. Second, the test procedures must be carried out repeatedly once for every system generation (e.g., system build) to ensure that security errors have not been introduced during system maintenance. Repeated manual performance of the entire

test suite may become a time consuming, error-prone activity. Third, availability of automated test suites enables evaluators to verify both the quality and extent of a vendor's test suite on an installed system in an expeditious manner. This significantly reduces the time required to evaluate that vendor's test suite.

The automation of most test procedures depends to a certain extent on the nature of the TCB interface under test. For example, for most TCB-primitive tests that require the same type of login, file system and directory initialization, it is possible to automate the tests by grouping test procedures in one or several user-level processes that are initiated by a single test-operator login. However, some TCB interfaces, such as the login and password change interfaces, must be tested from a user and administrator terminal. Similarly, the testing of the TCB interface primitives of B2 to A1 systems available to users only through trusted-path invocation requires terminal interaction with the test operator. Whenever security testing requires terminal interaction, test automation becomes a challenging objective.

Different approaches to test automation are possible. First, test designers may want to separate test procedures requiring terminal interaction (which are not usually automated), from those that do not require terminal interaction (which are readily amenable to automation). In this approach, the minimization of the number of test procedures that require terminal interaction is recommended.

Second, when test procedures requiring human-operator interaction cannot be avoided, test designers may want to connect a workstation to a terminal line and simulate the terminal activity of a human test operator on the workstation. This enables the complete automation of the test environment initialization and execution procedures, but not necessarily of the result identification and analysis procedure. This approach has been used in the testing of the Secure XenixTM ¹ TCB. The commands issued by the test workstation that simulates the human-operator commands are illustrated in the appendix of reference [9].

Third, the expected outcome of each test should be represented in the same format as that assumed by the output of the TCB under test and should be placed in files of the workstation simulating a human test operator. The comparison between the outcome files and the test result files (transferred to the workstation upon test

1. Secure Xenix is supported by Trusted Information Systems, Inc. as Trusted Xenix.

completion) can be performed using simple tools for file comparisons available in most current operating systems. The formatting of the outcome files in a way that allows their direct comparison with the test program output is a complex process. In practice, the order of the outcomes is determined only at the time the test programs are written, and sometimes only at execution time. Automated analysis of test results is seldomly done for this reason. To aid analysis of test results by human operators, the test result outputs can label and time-stamp each test. Intervention by a human test operator is also necessary in any case of mismatches between obtained test results and expected outcomes.

An approach to automating security testing using Prolog is presented in reference [20].

3.3 TESTING APPROACHES

All approaches to security functional testing require the following four major steps: (1) the development of test plans (i.e., test conditions, test data including test outcomes, and test coverage analysis) and execution for each TCB primitive, (2) the definition of test procedures, (3) the development of test programs, and (4) the analysis of the test results. These steps are not independent of each other in all methods. Depending upon how these steps are performed in the context of security testing, three approaches can be identified: the monolithic (black-box) testing approach, the functional-synthesis (white-box) testing approach, and a combination of the two approaches called the gray-box testing approach.

In all approaches, the functions to be tested are the security-relevant functions of each TCB primitive that are visible to the TCB interface. The definition of these security functions is given by:

Classes C1 and C2. System documentation defining a system protection philosophy, mechanisms, and system interface operations (e.g., system calls).

Class B1. Informal interpretation of the (informal) security model and the system documentation.

Classes B2 and B3. Descriptive Top-Level Specifications (DTLSs) of the TCB and by the interpretation of the security model that is supposed to be implemented by the TCB functions.

Class A1. Formal Top-Level Specifications (FTLSs) of the TCB and by the interpretation of the security model that is supposed to be implemented by the TCB functions.

Thus, a definition of the correct security function exists for each TCB primitive of a system designed for a given security class. In TCB testing, major distinctions between the approaches discussed in the previous section appear in the areas of test plan generation (i.e., test condition, test data, and test coverage analysis). Further distinctions appear in the ability to eliminate redundant TCB-primitive tests without loss of coverage. This is important for TCB testing because a large number of access checks and access check sequences performed by TCB kernels are shared between different kernel primitives.

3.3.1 Monolithic (Black-Box) Testing

The application of the monolithic testing approach to TCBs and to trusted processes is outlined in reference [2]. The salient features of this approach to TCB testing are the following: (1) the test condition selection is based on the *TCSEC* requirements and include discretionary and mandatory security, object reuse, labeling, accountability, and TCB isolation; (2) the test conditions for each TCB primitive should be generated from the chosen interpretation of each security function and primitive as defined above (for each security class). Very seldom is the relationship between the model interpretation and the generated test conditions, data, and programs shown explicitly [3 and 4]. Without such a relationship, it is difficult to argue coherently that all relevant security features of the given system are covered.

The test data selection must ensure test environment independence for unrelated tests or groups of tests (e.g., discretionary vs. mandatory tests). Environment independence requires, for example, that the subjects, objects, and access privileges used in unrelated tests or groups of tests must differ in all other tests or group of tests.

The test coverage analysis, which usually determines the extent of the testing for any TCB primitive, is used to delimit the number of test sets and programs. In the monolithic approach, the test data is usually chosen by boundary-value analysis. The test data places the test program directly above, or below, the extremes of a set of equivalent inputs and outputs. For example, a boundary is tested in the case of the "read" TCB call to a file by showing that (1) whenever a user has the read privilege for that file, the read TCB call succeeds; and (2) whenever the read privilege for that file is revoked, or whenever the file does not exist, the read TCB call fails. Similarly, a boundary is tested in the case of TCB-call parameter validation by showing that a TCB call with parameters passed by reference (1) succeeds whenever the reference points to an object in the caller's address space, and (2) fails whenever the reference points to an object in another address space (e.g., kernel space or other user spaces).

To test an individual boundary condition, all other related boundary conditions must be satisfied. For example, in the case of the "read" primitive above, the test call must not try to read beyond the limit of a file since the success/failure of not reading/reading beyond this limit represents a different, albeit related, boundary condition. The number of individual boundary tests for N related boundary conditions is of the order $2N$ (since both successes and failures must be tested for each of the N conditions). Some examples of boundary-value analysis are provided in [2] for security testing, and in [5] and [6] for security-unrelated functional testing.

The monolithic testing approach has a number of practical advantages. It can always be used by both implementors and users (evaluators) of TCBs. No specific knowledge of implementation details is required because there is no requirement to break the TCB (e.g., kernel) isolation or to circumvent the TCB protection mechanism (to read, modify, or add to TCB code). Consequently, no special tools for performing monolithic testing are required. This is particularly useful in processor hardware testing when only descriptions of hardware/firmware-implemented instructions, but no internal hardware/firmware design documents, are available.

The disadvantages of the monolithic approach are apparent. First, it is difficult to provide a precise coverage assessment for a set of TCB-primitive tests, even though the test selection may cover the entire set of security features of the system. However, no coverage technique other than boundary-value analysis can be more

adequate without TCB code analysis. Second, the elimination of redundant TCB-primitive tests without loss of coverage is possible only to a limited extent; i.e., in the case of access-check dependencies (discussed below) among TCB-primitive specifications. Third, in the context of TCB testing, the monolithic approach cannot cope with the problem of cyclic dependencies among test programs. Fourth, lack of TCB-code analysis precludes the possibility of distinguishing between design and implementation code errors in all but a few special cases. Also, it precludes the discovery of spurious code within the TCB—a necessary condition for Trojan Horse analysis.

In spite of these disadvantages, monolithic functional testing can be applied successfully to TCB primitives that implement simple security checks and share few of these checks (i.e., few or no redundant tests would exist). For example, many trusted processes have these characteristics, and thus this approach is adequate.

3.3.2 Functional-Synthesis (White-Box) Testing

Functional-synthesis-based testing requires the test of both functions implemented by each program (e.g., program of a TCB primitive) as a whole and functions implemented by internal parts of the program. The internal program parts correspond to the functional ideas used in building the program. Different forms of testing procedures are used depending upon different kinds of functional synthesis (e.g., control, algebraic, conditional, and iterative synthesis described in [1] and [7]). As pointed out in [9], only the control synthesis approach to functional testing is suitable for security testing.

In control synthesis, functions are represented as sequences of other functions. Each function in a sequence transforms an input state into an output state, which may be the input to another function. Thus, a control synthesis graph is developed during program development and integration with nodes representing data states and arcs representing state transition functions. The data states are defined by the variables used in the program and represent the input to the state transition functions. The assignment of program functions, procedures, and subroutines to the state transition functions of the graph is usually left to the individual programmer's judgment. Examples of how the control synthesis graphs are built during the program development and integration phase are given in [1] and [7].

The suitability of the control synthesis approach to TCB testing becomes apparent when one identifies the nodes of the control synthesis graph with the access checks within the TCB and the arcs with data states and outcomes of previous access checks. This representation, which is the dual of the traditional control synthesis graphs [9], produces a kernel access-check graph (ACG). This representation is useful because in TCB testing the primary access-check concerns are those of (1) missing checks within a sequence of required checks, (2) wrong sequences of checks, and (3) faulty or incomplete access checks. (Many of the security problems identified in the Multics kernel design project existed because of these broad categories of inadequate access checks [8].) It is more suitable than the traditional control-synthesis graph because major portions of a TCB, namely the kernel, have comparatively few distinct access checks (and access-check sequences) and a large number of object types and access privileges that have the same access-check sequences for different TCB primitives [9]. (However, this approach is less advantageous in trusted process testing because trusted processes—unlike kernels—have many different access checks and few shared access sequences.) These objects cause the same data flow between access check functions and, therefore, are combined as graph arcs.

The above representation of the control synthesis graph has the advantage of allowing the reduction of the graph to the subset of kernel functions that are relevant to security testing. In contrast, a traditional graph would include (1) a large number of other functions (and, therefore, graph arcs), and (2) a large number of data states (and, therefore, graph nodes). This would be both inadequate and unnecessary. It would be inadequate because the presence of a large number of security-irrelevant functions (e.g., functions unrelated to security or accountability checks or to protection mechanisms) would obscure the role of the security-relevant ones, making test coverage analysis a complex and difficult task. It would be unnecessary because not only could security-irrelevant functions be eliminated from the graph but also the flows of different object types into the same access check function could be combined, making most object type-based security tests unnecessary.

Any TCB-primitive program can be synthesized at the time of TCB implementations as a graph of access-checking functions and data flow arcs. Many of the TCB-primitive programs share both arcs and nodes of the TCB graph. To build an access-check graph, one must identify all access-check functions, their

inputs and outputs, and their sequencing. A typical input to an access-check function consists of an object identifier, object type and required access privileges. The output consists of the input to the next function (as defined above) and, in most cases, the outcome of the function check. The sequencing information for access-check functions consists of (1) the ordering of these functions, and (2) the number of arc traversals for each arc. An example of this is the sequencing of some access check functions that depend on the object types.

Test condition selection in the control-synthesis approach can be performed so that all the above access check concerns are satisfied. For example, test conditions must identify missing discretionary, mandatory, object reuse, privilege-call, and parameter validation checks (or parts of those checks). It also must identify access checks that are out of order, and faulty or incomplete checks, such as being able to truncate a file for which the modify privilege does not exist. The test conditions must also be based on the security model interpretation to the same extent as that in the monolithic approach.

The test coverage in this approach also refers to the delimitation of the test data and programs for each TCB primitive. Because many of the access-check functions, and sequences of functions, are common to many of the kernel primitives (but not necessarily to trusted-process primitives), the synthesized kernel (TCB) graph is fairly small. Despite this the coverage analysis cannot rely on individual arc testing for covering the graph. The reason is that arc testing does not force the testing of access checks that correspond to combinations of arcs and thus it does not force coverage of all relevant sequences of security tests. Newer test coverage techniques for control synthesis graphs, such as data-flow testing [9, 10, and 11] provide coverage of arc combinations and thus are more appropriate than those using individual arc testing.

The properties of the functional-synthesis approach to TCB testing appear to be orthogonal to those of monolithic testing. Consider the disadvantages of functional-synthesis testing. It is not as readily usable as monolithic testing because of the lack of detailed knowledge of system internals. Also, it helps remove very few redundant tests whenever few access check sequences are shared by TCB primitives (as is the case with most trusted-process primitives).

Functional-synthesis-based testing, however, has a number of fundamental advantages. First, the coverage based on knowledge of internal program structure (i.e., code structure of a kernel primitive) can be more extensive than in the monolithic approach [1 and 7]. A fairly precise assessment of coverage can be made, and most of the redundant tests can be identified. Second, one can distinguish between TCB-primitive program failures and TCB-primitive design failures, something nearly impossible with monolithic testing. Third, this approach can help remove cyclic test dependencies. By removing all, or a large number of redundant tests, one removes most cyclic test dependencies (example of Section 3.7.5).

TCB code analysis becomes necessary whenever a graph synthesis is done after a TCB is built. Such analysis helps identify spurious control paths and code within a TCB—a necessary condition for Trojan Horse discovery. (In such a case, a better term for this approach would be functional-analysis-based testing.)

3.3.3 Gray-Box Testing

Two of the principal goals of security testing have been (1) the elimination of redundant tests through systematic test-condition selection and coverage analysis, and (2) the elimination of cyclic dependencies between the test programs. Other goals, such as test repeatability, which is also considered important, can be attained through the same means as those used for the other methods.

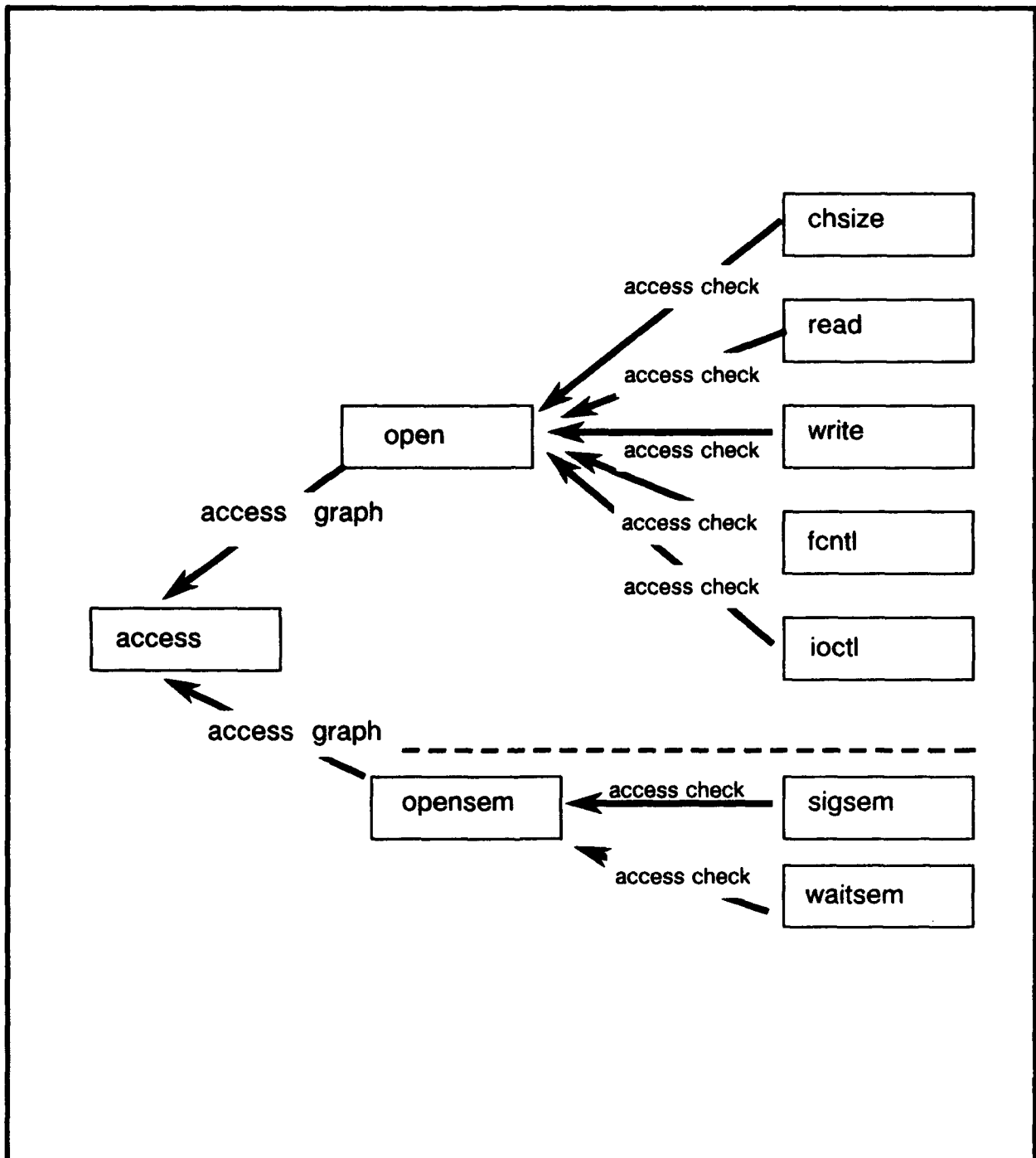
The elimination of redundant TCB-primitive tests is a worthwhile goal for the obvious reason that it reduces the amount of testing effort without loss of coverage. This allows one to determine a smaller nucleus of tests that must be carried out extensively. The overall TCB assurance may increase due to the judicious distribution of the test effort. The elimination of cyclic dependencies among the TCB-primitive test programs is also a necessary goal because it helps establish a rigorous test order without making circular assumptions of the behavior of the TCB primitives. Added assurance is therefore gained.

To achieve the above goals, the gray-box testing approach combines monolithic testing with functional-synthesis-based testing in the test selection and coverage areas. This combination relies on the elimination of redundant tests through access-check dependency analysis afforded by monolithic testing. It also relies on

the synthesis of the access-check graph from the TCB code as suggested by functional-synthesis-based testing (used for further elimination of redundant tests). The combination of these two testing methods generates a TCB-primitive test order that requires increasingly fewer test conditions and data without loss of coverage.

A significant number of test conditions and associated tests can be eliminated by the use of the access-check graph of TCB kernels. Recall that each kernel primitive may have a different access-check graph in principle. In practice, however, substantial parts of the graphs overlap. Consequently, if one of the graph paths is tested with sufficient coverage for a kernel primitive, then test conditions generated for a different kernel primitive whose graph overlaps with the first need only include the access checks specific to the latter kernel primitive. This is true because by the definition of the access-check graph, the commonality of paths means that the same access checks are performed in the same sequence, on the same types of objects and privileges, and with the same outcomes (e.g., success and failure returns). The specific access checks of a kernel primitive, however, must also show that the untested subpath(s) that has not been tested, of that kernel primitive, joins the tested path.

(A subset of the access-check and access-graph dependencies for the **access**, **open**, **read**, **write**, **fcntl**, **ioctl**, **opensem**, **waitsem** and **sigsem** primitives of UnixTM-like kernels are illustrated in Figures 1 and 2, pages 23 and 24. The use of these dependencies in the development of test plans, especially in coverage analysis, is illustrated in Sections 3.7.2.3 and 3.7.3.3; namely, in the test plans for **access**, **open**, and **read**. Note that the arcs shown in Figure 2, page 24 include neither complete flow-of-control information nor complete sets of object types, access-checks per call, and call outcome.)

Examples of Access-Graph and Access-Check Dependencies in Unix™**Figure 1**

Example of Access-Check Graph for Three Kernel Calls

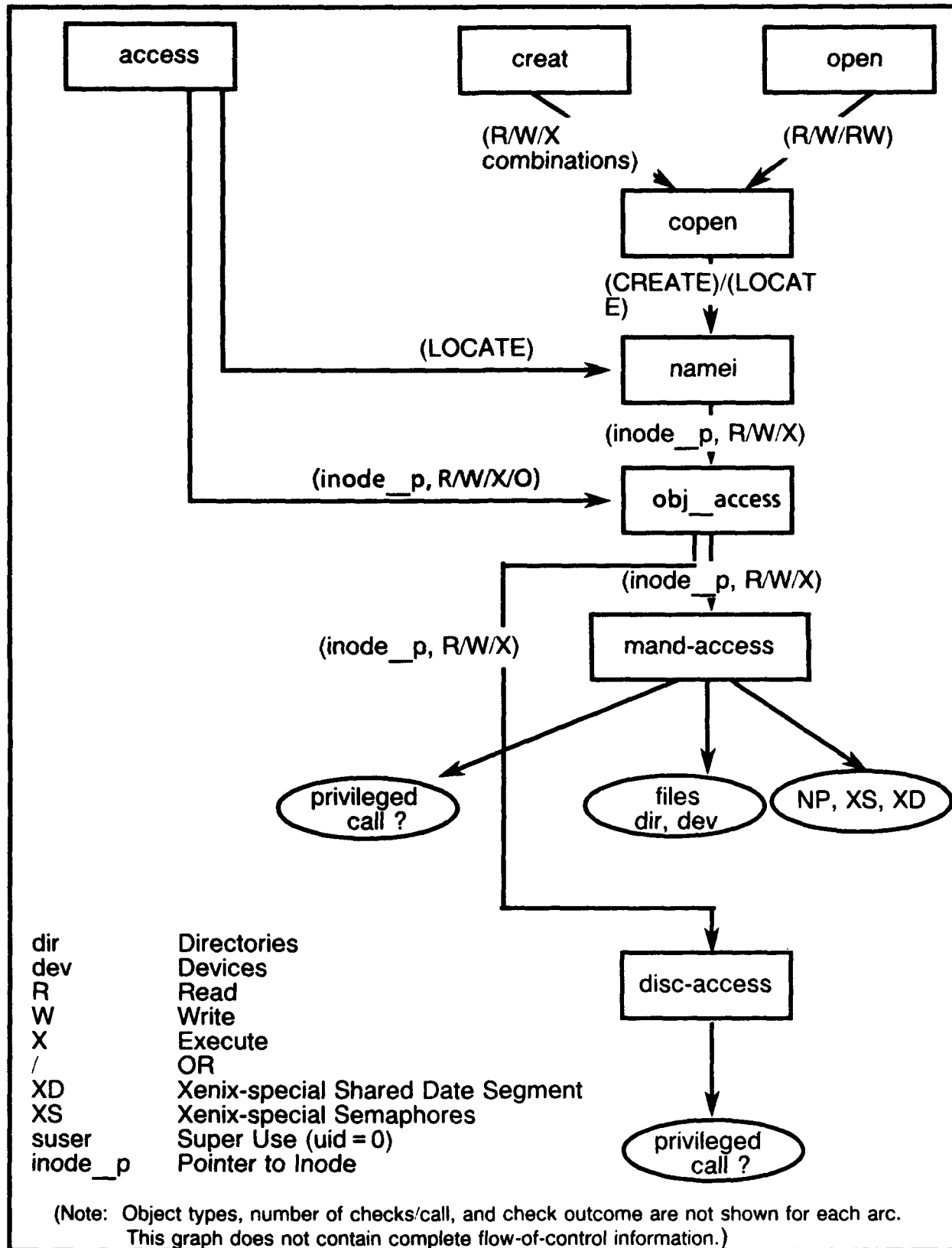


Figure 2

3.4 RELATIONSHIP WITH THE TCSEC SECURITY TESTING REQUIREMENTS

The TCSEC security testing requirements and guidelines (i.e., Part 1 and Section 10 of the TCSEC) help define different approaches for security testing. They are particularly useful for test condition generation and test coverage. This section reviews these requirements in light of security testing approaches defined in Section 3.3.

Security Class C1

Test Condition Generation

"The security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation."
[TCSEC Part 1, Section 2.1]

For this class of systems, the test conditions should be generated from the system documentation which includes the Security Features User's Guide (SFUG), the Trusted Facility Manual (TFM), the system reference manual describing each TCB primitive, and the design documentation defining the protection philosophy and its TCB implementation. Both the SFUG and the manual pages, for example, illustrate how the identification and authentication mechanisms work and whether a particular TCB primitive contains relevant security and accountability mechanisms. The Discretionary Access Control (DAC) and the identification and authentication conditions enforced by each primitive (if any) are used to define the test conditions of the test plans.

Test Coverage

"Testing shall be done to assure that there are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCB." [TCSEC, Part I, Section 2.1]

"The team shall independently design and implement at least five system-specific tests in an attempt to circumvent the security mechanisms of the system." [TCSEC, Part II, Section 10]

The above TCSEC requirements and guidelines define the scope of security testing for this security class. Since each TCB primitive may include security-relevant mechanisms, security testing shall include at least five test conditions for each primitive. Furthermore, because source code analysis is neither required nor suggested for class C1 systems, monolithic functional testing (i.e., a black-box

approach) with boundary-value coverage represents an adequate testing approach for this class. Boundary-value coverage of each test condition requires that at least two calls of each TCB primitive be made, one for the positive and one for the negative outcome of the condition. Such coverage may also require more than two calls per condition. Whenever a TCB primitive refers to multiple types of objects, each condition is repeated for each relevant type of object for both its positive and negative outcomes. A large number of test calls may be necessary for each TCB primitive because each test condition may in fact have multiple related conditions which should be tested independently of each other.

Security Class C2

Test Condition Generation

"Testing shall also include a search for obvious flaws that would allow violation of resource isolation, or that would permit unauthorized access to the audit and authentication data." [TCSEC, Part 1, Section 2.2]

These added requirements refer only to new sources of test conditions, but not to a new testing approach nor to new coverage methods. The following new sources of test conditions should be considered:

- (1) Resource isolation conditions. These test conditions refer to all TCB primitives that implement specific system resources (e.g., object types or system services). Test conditions for TCB primitives implementing services may differ from those for TCB primitives implementing different types of objects. Thus, new conditions may need to be generated for TCB services. The mere repetition of test conditions defined for other TCB primitives may not be adequate for some services.
- (2) Conditions for protection of audit and authentication data. Because both audit and authentication mechanisms and data are protected by the TCB, the test conditions for the protection of these mechanisms and their data are similar to those which show that the TCB protection mechanisms are tamperproof and noncircumventable. For example, these conditions show that neither privileged TCB primitives nor audit and user authentication files are accessible to regular users.

Test Coverage

Although class C1 test coverage already suggests that each test condition be covered for each type of object, coverage of resource-specific test conditions also requires that each test condition be covered for each type of service (whenever the test condition is relevant to a service). For example, the test conditions which show that direct access to a shared printer is denied to a user shall be repeated for a shared tape drive with appropriate modification of test data (i.e., test environments set up, test parameters and outcomes—namely, the test plan structure discussed in Section 3.5).

Security Class B1

Test Condition Generation

The objectives of security testing “. . . shall be: to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy enforced by the TCB; as well as to ensure that no subject (without authorization to do so) is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users.” [TCSEC, Part 1, Section 3.1]

The security testing requirements of class B1 are more extensive than those of both classes C1 and C2, both in test condition generation and in coverage analysis. The source of test conditions referring to users' access to data includes the mandatory and discretionary policies implemented by the TCB. These policies are defined by an (informal) policy model whose interpretation within the TCB allows the derivation of test conditions for each TCB primitive. Although not explicitly stated in the *TCSEC*, it is generally expected that all relevant test conditions for classes C1 and C2 also would be used for a class B1 system.

Test Coverage

“All discovered flaws shall be removed or neutralized and the TCB retested to demonstrate that they have been eliminated and that new flaws have not been introduced.” [TCSEC, Part 1, Section 3.1]

“The team shall independently design and implement at least fifteen system specific tests in an attempt to circumvent the security mechanisms of the system.” [TCSEC, Part II, Section 10]

Although the coverage analysis is still boundary-value analysis, security testing for class B1 systems suggests that at least fifteen test conditions be generated for each TCB primitive that contains security-relevant mechanisms to cover both mandatory and discretionary policy. In practice, however, a substantially higher number of test conditions is generated from interpretations of the (informal) security model. The removal or the neutralization of found errors and the retesting of the TCB requires no additional types of coverage analysis.

Security Class B2

Test Condition Generation

"Testing shall demonstrate that the TCB implementation is consistent with the descriptive top-level specification." [TCSEC, Part 1, Section 3.2]

The above requirement implies that both the test conditions and coverage analysis of class B2 systems are more extensive than those of class B1. In class B2 systems every access control and accountability mechanism documented in the DTLS (which must be complete as well as accurate) represents a source of test conditions. In principle the same types of test conditions would be generated for class B2 systems as for class B1 systems, because (1) in both classes the test conditions could be generated from interpretations of the security policy model (informal at B1 and formal at B2), and (2) in class B2 the DTLS includes precisely the interpretation of the security policy model. In practice this is not the case however, because security policy models do not model a substantial number of mechanisms that are, nevertheless, included in the DTLS of class B2 systems. (Recall that class B1 systems do not require a DTLS of the TCB interface.) The number and type of test conditions can therefore be substantially higher in a class B2 system than those in a class B1 system because the DTLS for each TCB primitive may contain additional types of mechanisms, such as those for trusted facility management.

Test Coverage

It is not unusual to have a few individual test conditions for at least some of the TCB primitives. As suggested in the gray-box approach defined in the previous section, repeating these conditions for many of the TCB primitives to achieve uniform coverage can be both impractical and unnecessary. Particularly this is true

when these primitives refer to the same object types and services. It is for this reason and because source-code analysis is required in class B2 systems to satisfy other requirements that the use of the gray-box testing approach is recommended for the parts of the TCB in which primitives share a substantial portion of their code. Note that the DTLS of any system does not necessarily provide any test conditions for demonstrating the tamperproofness and noncircumventability of the TCB. Such conditions should be generated separately.

Security Class B3

Test Condition Generation

The only difference between classes B2 and B3 requirements of security testing reflects the need to discover virtually all security policy flaws before the evaluation team conducts its security testing exercise. Thus, no additional test condition requirements appear for class B3 testing. Note that the DTLS does not necessarily provide any test conditions for demonstrating the TCB is tamperproof and noncircumventable as with class B2 systems. Such conditions should be generated separately.

Test Coverage

“No design flaws and no more than a few correctable implementation flaws may be found during testing and there shall be reasonable confidence that few remain.” [TCSEC, Part 1, Section 3.3]

The above requirement suggests that a higher degree of confidence in coverage analysis is required for class B3 systems than for class B2 systems. It is for this reason that it is recommended the gray-box testing approach be used extensively for the entire TCB kernel, and data-flow coverage be used for all independent primitives of the kernel (namely, the gray-box method in Section 3.3 above).

Security Class A1

The only differences between security testing requirements of classes B3 and A1 are (1) the test conditions shall be derived from the FTLS, and (2) the coverage analysis should include at least twenty-five test conditions for each TCB primitive implementing security functions. Neither requirement suggests that a different testing method than that recommended for class B3 systems is required.

3.5 SECURITY TEST DOCUMENTATION

This section discusses the structure of typical test plans, test logs, test programs, test procedures, and test reports. The description of the test procedures necessary to run the tests and to examine the test results is also addressed. The documentation structures presented are meant to provide the system developers with examples of good test documentation.

3.5.1 Overview

The work plan for system testing should describe how security testing will be conducted and should contain the following information:

- Test-system configuration for both hardware and software.
- Summary test requirements.
- Procedures for executing test cases.
- Step-by-step procedures for each test case.
- Expected results for each test step.
- Procedures for correcting flaws uncovered during testing.
- Expected audit information generated by each test case (if any).

See Section 3.7.7, "Relationship with the TCSEC Requirements."

3.5.2 Test Plan

Analysis and testing of mechanisms, assurances and/or documentation to support the TCSEC security testing requirements are accomplished through test plans. The test plans should be sufficiently complete to cover each identified security mechanism and should be conducted with sufficient depth to provide reasonable assurance that any bugs not found lie within the acceptable risk threshold for the class of the system being evaluated. A test plan consists of test conditions, test data, and coverage analysis.

3.5.2.1 Test Conditions

A test condition is a statement of a security-relevant constraint that must be satisfied by a TCB primitive. Test conditions should be derived from the system's DTLS/FTLS, from the interpretation of the security and accountability models (if any), from TCB isolation and noncircumventability properties, and from the specifications and implementation of the individual TCB primitive under test. If neither DTLS/FTLS nor models are required, then test conditions should be derived from the informal policy statements, protection philosophy and resource isolation requirements.

(1) Generation of Model- or Policy-Relevant Test Conditions

This step suggests that a matrix of TCB primitives and the security model(s) or requirement components be built. Each entry in the matrix identifies the security relevance of each primitive (if any) in a security model or requirement area and the relevant test conditions. For example, in the mandatory access control area of security policy, one should test the proper object labeling by the TCB, the "compatibility" property of the user created objects, and the TCB implemented authorization rules for subject access to objects. One should also test that the security-level relationships are properly maintained by the TCB and that the mandatory access works independently of, and in conjunction with, the discretionary access control mechanism. In the discretionary access control area, one may include tests for proper user/group identifier selection, proper user inclusion/exclusion, selective access distribution/revocation using the access control list (ACL) mechanism, and access review.

Test conditions derived from TCB isolation and noncircumventability properties include conditions that verify (1) that TCB data structures are inaccessible to user level programs, (2) that transfer of control to the TCB can take place only at specified entry points, which cannot be bypassed by user-level programs, (3) that privileged entry points into the TCB cannot be used by user level programs, and (4) that parameters passed by reference to the TCB are validated.

Test conditions derived from accountability policy include conditions that verify that user identification and authentication mechanisms operate properly. For example, they include conditions that verify that only sufficiently complex passwords can be chosen by any user, that the password aging mechanism forces reuse at

stated intervals, and so on. Other conditions of identification and authentication, such as those that verify that the user login level is dominated by the user's maximum security level, should also be included. Furthermore, conditions that verify that the user commands included in the trusted path mechanism are unavailable to the user program interface of the TCB should be used. Accountability test conditions that verify the correct operation of the audit mechanisms should also be generated and used in security testing.

The security relevance of a TCB primitive can only be determined from the security policy, accountability, and TCB isolation and noncircumventability requirements for classes B1 to A1, or from protection philosophy and resource isolation requirements for classes C1 and C2. Some TCB primitives are security irrelevant. For example, TCB primitives that never allow the flow of information across the boundaries of an accessible object are always security irrelevant and need not be tested with respect to the security or accountability policies. The limitation of information flow to user-accessible objects by the TCB primitives implementation, however, needs to be tested by TCB-primitive-specific tests. A general example of security-irrelevant TCB primitives is provided by those primitives which merely retrieve the status of user-owned processes at the security level of the user.

(2) Generation of TCB-Primitive-Specific Test Conditions

The selection of test conditions used in security testing should be TCB-primitive-specific. This helps remove redundant test conditions and, at the same time, helps ensure that significant test coverage is obtained. For example, the analysis of TCB-primitive specifications to determine their access-check dependencies is required whenever the removal of redundant TCB-primitive tests is considered important. This analysis can be applied to all testing approaches. The specification of a TCB primitive A is access-check dependent on the specification of a TCB primitive B if a subset of the access checks needed in TCB primitive A are performed in TCB primitive B, and if a TCB call to primitive B always precedes a TCB call to primitive A (i.e., a call to TCB primitive A fails if the call to TCB primitive B has not been done or has not completed with a successful outcome). In case of such dependencies, it is sufficient to test TCB primitive B first and then to test only the access checks of

TCB primitive A that are not performed in TCB primitive B. Of course, the existence of the access-check dependency must be verified through testing.

As an example of access-check dependency, consider the **fork** and the **exit** primitives of the Secure Xenix™ kernel. The **exit** primitive always terminates a process and sends a return code to the parent process. The mandatory access check that needs to be tested in **exit** is that the child's process security level equals that of the parent's process. However, the specifications of the **exit** primitive are access-check dependent on the specifications of the **fork** primitive (1) because an **exit** call succeeds only after a successfully completed **fork** call is done by some parent process, and (2) because the access check, that the child's process level always equals that of the parent's process level, is already performed during the **fork** call. In this case, no additional mandatory access test is needed for **exit** beyond that performed for **fork**. Similarly, the **sigsem** and the **waitsem** primitives of some Unix™-based kernels are access-check dependent on the **opensem** primitive, and no additional mandatory or discretionary access checks are necessary.

However, in the case of the **read** and the **write** primitives of Unix™ kernels, the specifications of which are also access-check dependent on both the mandatory and the discretionary checks of the **open** primitive, additional tests are necessary beyond those done for **open**. In the case of the **read** primitive one needs to test that files could only be read if they have been opened for reading, and that reading beyond the end of a file is impossible after one tests the dependency of **read** on the specification of **open**. Additional tests are also needed for other primitives such as **fcntl** and **ioctl**; their specifications are both mandatory and discretionary access-check dependent on the **open** primitives for files and devices. Note that in all of the above examples a large number of test conditions and associated tests are eliminated by using the notion of access check dependency of specifications because, in general, less test conditions are generated for access check dependency testing than for the security testing of the primitive itself.

The following examples are given in references [3] and [4]: (1) of the generation of such constraints from security models, (2) of the predicates, variables, and object types used in constraint definition, and (3) of the use of such constraints in test conditions for processor instructions (rather than for TCB primitives).

See Section 3.7.7, "Relationship with the TCSEC Requirements."

3.5.2.2 Test Data

"Test data" is defined as the set of specific objects and variables that must be used to demonstrate that a test condition is satisfied by a TCB primitive. The test data consist of the definition of the initialization data for the test environment, the test parameters for each TCB primitive, and the expected test outcomes. Test data generation is as important as test condition generation because it ensures that test conditions are exercised with appropriate coverage in the test programs, and that test environment independence is established whenever it is needed.

To understand the importance of test data generation consider the following example. Suppose that all mandatory tests must ensure that the "hierarchy" requirement of the mandatory policy interpretation must be tested for each TCB primitive. (Expansion on this subject, i.e., the nondecreasing security level requirement for the directory hierarchy can be found in [12].) What directory hierarchy should one set up for testing this requirement and at the same time argue that all possible directory hierarchies are covered for all tests? A simple analysis of this case shows that there are two different forms of upgraded directory creation that constitute an independent basis for all directory hierarchies (i.e., all hierarchies can be constructed by the operations used for one or the other of the two forms, or by combinations of these operations). The first form is illustrated in Figure 3a representing the case whereby each upgraded directory at a different level is upgraded from a single lower level (e.g., system low). The second form is illustrated in Figure 3b and represents the case whereby each directory at a certain level is upgraded from an immediately lower level. A similar example can be constructed to show that combinations of security level definitions used for mandatory policy testing cover all security level relationships.

Test data for TCB primitives should include several items such as the TCB primitive input data, TCB primitive return result and success/failure code, object hierarchy definition, security level used for each process/object, access privileges used, user identifiers, object types, and so on. This selection needs to be made on a test-by-test basis and on a primitive-by-primitive basis. Whenever environment independence is required, a different set of data is defined [2]. It is very helpful that the naming scheme used for each data object helps identify the test that used that

Examples of Different Directory Hierarchies
All Directories are Upgraded From the Same "home" Directory

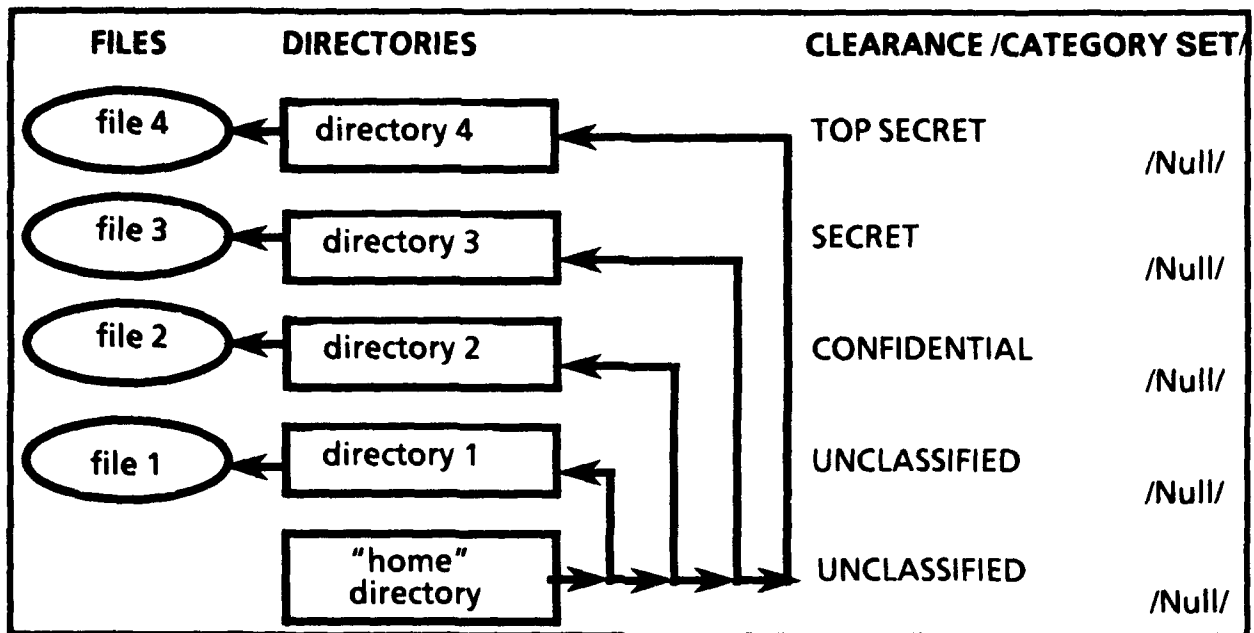


Figure 3a

Examples of Different Directory Hierarchies
All Directories are Upgraded Sequence

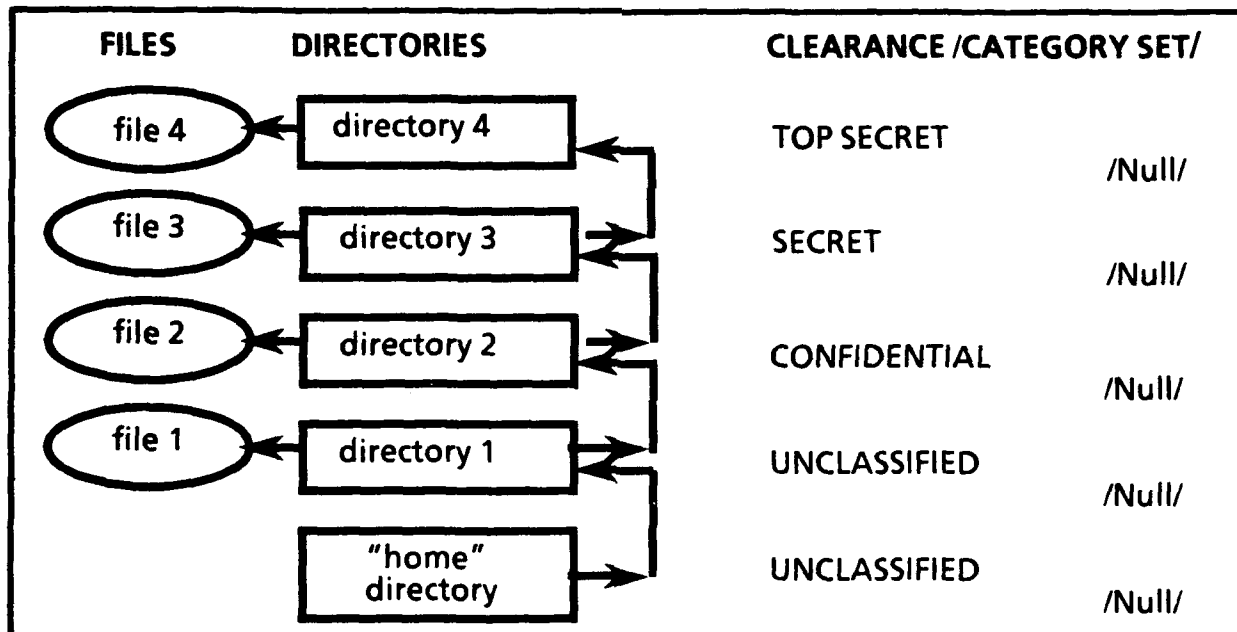


Figure 3b

item. Different test environments can be easily identified in this way. Note that the test data selection should ensure both coverage of model-relevant test conditions and coverage of the individual TCB primitives. This will be illustrated in an example in the next section.

See Section 3.7.7, "Relationship with the TCSEC Requirements."

3.5.2.3 Coverage Analysis

Test coverage analysis is performed in conjunction with the test selection phase of our approach. Two classes of coverage analysis should be performed: model- or policy-dependent coverage and individual TCB primitive coverage.

(1) Model- or Policy-Dependent Coverage

In this class, one should demonstrate that the selected test conditions and data cover the interpretation of the security and accountability model and noncircumventability properties in all areas identified by the matrix mentioned above. This is a comparatively simple task because model coverage considerations drive the test condition and data selection. This kind of coverage includes object type, object hierarchy, subject identification, access privilege, subject/object security level, authorization check coverage, and so on. Model dependent coverage analysis relies, in general, on boundary-value analysis.

(2) Individual TCB-Primitives Coverage

This kind of coverage includes boundary value analysis, data flow analysis of individual access-check graphs of TCB primitives, and coverage of dependencies. The examples of reference [2] illustrate boundary-value analysis. Other forms of TCB-primitive coverage will be discussed in Section 3.7 of this guideline. For example, graph coverage analysis represents the determination that the test conditions and data exercise all the data flows for each TCB-primitive graph. This includes not only the traversal of all the graph access checks (i.e., nodes) but also of all the graph's arcs and arc sequences required for each TCB primitive. (The example for **access** primitive of UnixTM kernels included in Section 3.7 explains this form of coverage. Data flow coverage is also presented in [10] and [11] for security-unrelated test examples.)

Coverage analysis is both a qualitative and quantitative assessment of the extent to which the test shows TCB-primitive compliance with the (1) design documentation, (2) resource isolation, (3) audit and authentication data protection, (4) security policy and accountability model conditions, (5) DTLS/FTLS, as well as with those of the TCB isolation and noncircumventability properties. To achieve significant coverage, all security-relevant conditions derived from a TCB model and properties and DTLS/FTLS should be covered by a test, and each TCB-primitive test should cover the implementation of its TCB primitive. For example, each TCB-primitive test should be performed for all independent object types operated upon by that TCB primitive and should test all independent security exceptions for each type of object.

See Section 3.7.7, "Relationship with the TCSEC Requirements."

3.5.3 Test Procedures

A key step in any test system is the generation of the test procedures (which are also known as "test scripts"). The major function of the test procedure is to ensure that an independent test operator or user is able to carry out the test and to obtain the same results as the test implementor. The procedure for each test should be explained in sufficient detail to enable repeatable testing. The test procedure should contain the following items to accomplish this:

- (1) **Environment Initialization Procedure.** This procedure defines the login sequences and parameters, the commands for object and subject cleanup operations at all levels involved in the test, the choice of object names, the commands and parameters for object creation and initialization at the required levels, the required order of command execution, the initialization at the required levels, the initialization of different subject identifiers and access privileges (for the initialized objects) at all required levels, and the specification of the test program and command names and parameters used in the current test.
- (2) **Test Execution Procedure.** The test procedure includes a description of the test execution from a terminal including the list of user commands, their input, and the expected terminal, printer, or file output.
- (3) **Result Identification Procedure.** The test procedure should also identify the results file for a given test, or the criteria the test operator must use to find the

results of each individual test in the test output file. The meaning of the results should also be provided.

See Section 3.7.7, "Relationship with the TCSEC Requirements."

Note: A system in which testing is fully automated eliminates the need for separate test procedure documentation. In such cases, the environment initialization procedures and the test execution procedures should be documented in the test data section of the test plans. Automated test operator programs include the built-in knowledge otherwise contained in test procedures.

3.5.4 Test Programs

Another key step of any test system is the generation of the test programs. The test programs for each TCB primitive consist of the login sequence, password, and requested security level. The security profile of the test operator and of the possible workstation needs to be defined a priori by the system security administrators to allow logins and environment initialization at levels required in the test plan. After login, a test program invokes several trusted processes (e.g., "mkdir," "rmdir," in some UnixTM systems) with predetermined parameters in the test plan and procedure to initialize the test environment. A nucleus of trusted processes, necessary for the environment set up, are tested independently of a TCB primitive under test whenever possible and are assumed to be correct.

After the test environment is initialized, the test program (which may require multiple logins at different levels) issues multiple invocations to the TCB primitive under test and to other TCB primitives needed for the current test. The output of each primitive issued by the test programs is collected in a result file associated with each separate test and analyzed. The analysis of the test results that are collected in the results file is performed by the test operator. This analysis is a comparison between the results file and the expected outcome file defined by the test plan prior to the test run. Whenever the test operator detects a discrepancy between the two files he records a test error.

3.5.5 Test Log

A test log should be maintained by each team member during security testing. It is to capture useful information to be included later in the test report. The test log should contain:

- Information on any noteworthy observations.
- Modifications to the test steps.
- Documentation errors.
- Other useful data recorded during the testing procedure test results.

3.5.6 Test Report

The test report is to present the results of the security testing in a manner that effectively supports the conclusions reached from the security testing process and provides a basis for NCSC test team security testing. The test report should contain:

- Information on the configuration of the tested system.
- A chronology of the security testing effort.
- The results of functional testing including a discussion of each flaw uncovered.
- The results of penetration testing covering the results of successful penetrations.
- Discussion of the corrections that were implemented and of any retesting that was performed.

A sample test report format is provided in Section 3.7.

3.6 SECURITY TESTING OF PROCESSORS' HARDWARE/FIRMWARE PROTECTION MECHANISMS

The processors of a computer system include the Central Processing Units (CPU), Input/Output (I/O) processors, and application-oriented co-processors such as numerical co-processors and signal-analysis co-processors. These processors may include mechanisms capabilities, access privileges, processor-status registers,

and memory areas representing TCB internal objects such as process control blocks, descriptor, and page tables. The effects of the processor protection mechanisms become visible to the system users through the execution of processor instructions and I/O commands that produce transformations of processor and memory registers. Transformations produced by every instruction or I/O command are checked by the processors' protection mechanisms and are allowed only if they conform with the specifications defined by the processor reference manuals for that instruction. For few processors these transformations are specified formally and for less processors a formal (or informal) model of the protection mechanisms is given [3 and 4].

3.6.1 The Need for Hardware/Firmware Security Testing

Protection mechanisms of systems processors provide the basic support for TCB isolation, noncircumventability, and process address space separation. In general, processor mechanisms for the isolation of the TCB include those that (1) help separate the TCB address space and privileges from those of the user, (2) help enforce the transfer of control from the user address space to the TCB address space at specific entry points, and (3) help verify the validity of the user-level parameters passed to the TCB during primitive invocation. Processor mechanisms that support TCB noncircumventability include those that (1) check each object reference against a specific set of privileges, and (2) ensure that privileged instructions which can circumvent some of the protection mechanisms are inaccessible to the user. Protection mechanisms that help separate process address spaces include those using base and relocation registers, paging, segmentation, and combinations thereof.

The primary reason for testing the security function of a system's processors is that flaws in the design and implementation of processor-supported protection mechanisms become visible at the user level through the instruction set. This makes the entire system vulnerable because users can issue carefully constructed sequences of instructions that would compromise TCB and user security.

(User visibility of protection flaws in processor designs is particularly difficult to deny. Attempts to force programmers to use only high-level languages, such as PL1, Pascal, Algol, etc., which would obscure the processor instruction set, are counterproductive because arbitrary addressing patterns and instruction sequences

still can be constructed through seemingly valid programs (i.e., programs that compile correctly). In addition, exclusive reliance on language compilers and on other subsystems for the purpose of obscuring protection flaws and denying users the ability to produce arbitrary addressing patterns is unjustifiable. One reason is that compiler verification is a particularly difficult task; another is that reliance on compilers and on other subsystems implies reliance on the diverse skills and interests of system programmers. Alternatively, hardware-based attempts to detect instruction sequence patterns that lead to protection violations would only result in severe performance degradation.)

The additional reason for testing the security function of a system's processor is that, in general, a system's TCB uses at least some of the processor's mechanisms to implement its security policy. Flawed protection mechanisms may become unusable by the TCB and, in some cases, the TCB may not be able to neutralize those flaws (e.g., make them invisible to the user). It should be noted that the security testing of the processor protection mechanisms is the most basic life-cycle evidence available in the context of TCSEC evaluations to support the claim that a system's reference notion is verifiable.

3.6.2 Explicit TCSEC Requirements for Hardware Security Testing

The TCSEC imposes very few explicit requirements for the security testing of a system's hardware and firmware protection mechanisms. Few interpretations can be derived from these requirements as a consequence. Recommendations for processor test plan generation and documentation, however, will be made in this guideline in addition to explicit TCSEC requirements. These recommendations are based on analogous TCB testing recommendations made herein.

Specific Requirements for Classes C1 and C2

The following requirements are included for security classes C1 and C2:

"The security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation."

The security mechanisms of the ADP system clearly include the processor-supported protection mechanisms that are used by the TCB and those that are visible to the users through the processor's instruction set. In principle it could be argued that the TCB security testing implicitly tests at least some processor

mechanisms used by the TCB; therefore, no additional hardware testing is required for these mechanisms. All processor protection mechanisms that are visible to the user through the instruction set shall be tested separately regardless of their use by a tested TCB. In practice, nearly all processor protection mechanisms are visible to users through the instruction set. An exception is provided by some of the I/O processor mechanisms in systems where users cannot execute I/O commands either directly or indirectly.

Specific Requirements for Classes B1 to B3

In addition to the above requirements of classes C1 and C2, the *TCSEC* includes the following specific hardware security testing guidelines in Section 10 "A Guideline on Security Testing":

"The [evaluation] team shall have 'hands-on' involvement in an independent run of the test package used by the system developer to test security-relevant hardware and software."

The explicit inclusion of this requirement in the division B (i.e., classes B1 to B3) of the *TCSEC* guideline on security testing implies that the scope and coverage of the security-relevant hardware testing and test documentation should be consistent with those of the TCB security testing for this division. Thus, the security testing of the processor's protection mechanisms for division B systems should be more extensive than for division C (i.e., C1 and C2) systems.

Specific Requirements for Class A1

In addition to the requirements for divisions C and B, the *TCSEC* includes the following explicit requirements for hardware and/or firmware testing:

"Testing shall demonstrate that the TCB implementation is consistent with the formal top-level specifications." [Security Testing requirement] and

"The DTLS and FTLS shall include those components of the TCB that are implemented as hardware and/or firmware if their properties are visible at the TCB interface." [Design Specification and Verification requirement]

The above requirements suggest that all processor protection mechanisms that are visible at the TCB interface should be tested. The scope and coverage of the

security-relevant testing and test documentation should also be consistent with those of TCB security-relevant testing and test documentation for this division.

3.6.3 Hardware Security Testing vs. System Integrity Testing

Hardware security testing and system integrity testing differ in at least three fundamental ways. First, the scope of system integrity testing and that of hardware security testing is different. System integrity testing refers to the functional testing of the hardware/firmware components of a system including components that do not necessarily have a specific security function (i.e., do not include any protection mechanisms). Such components include the memory boards, busses, displays, adaptors for special devices, etc. Hardware security testing, in contrast, refers to hardware and firmware components that include protection mechanisms (e.g., CPU's and I/O processors). Failures of system components that do not include protection mechanisms may also affect system security just as they would affect reliability and system performance. Failures of components that include protection mechanisms can affect system security adversely. A direct consequence of the distinction between the scope of system integrity and hardware security testing is that security testing requirements vary with the security class of a system, whereas system integrity testing requirements do not.

Second, the time and frequency of system integrity and security testing are different. System integrity testing is performed periodically at the installation site of the equipment. System security testing is performed in most cases at component design and integration time. Seldom are hardware security test suites performed at the installation site.

Third, the responsibility for system integrity testing and hardware security testing is different. System integrity testing is performed by site administrators and vendor customer or field engineers. Hardware security testing is performed almost exclusively by manufacturers, vendors, and system evaluators.

3.6.4 Goals, Philosophy, and Approaches to Hardware Security Testing

Hardware security testing has the same general goals and philosophy as those of general TCB security testing. Hardware security testing should be performed for processors that operate in normal mode (as opposed to maintenance or test mode). Special probes, instrumentation, and special reserved op-codes in the instruction set

should be unnecessary. Coverage analysis for each tested instruction should be included in each test plan. Cyclic test dependencies should be minimized, and testing should be repeatable and automated whenever possible.

In principle, all the approaches to security testing presented in Section 3.3 are applicable to hardware security testing. In practice, however, all security testing approaches reported to date have relied on the monolithic testing approach. This is the case because hardware security testing is performed on an instruction basis (often only descriptions of the hardware/firmware-implemented, but no internal hardware/firmware design details, are available to the test designers). The generation of test conditions is, consequently, based on instruction and processor documentation (e.g., on reference manuals). Models of the processor protection mechanisms and top-level specifications of each processor instruction are seldom available despite their demonstrable usefulness [3 and 4] and mandatory use [13, class A1] in security testing. Coverage analysis is restricted in practice to boundary-value coverage for similar reasons.

3.6.5 Test Conditions, Data, and Coverage Analysis for Hardware Security Testing

Lack of DTLS and protection-model requirements for processors' hardware/firmware in the TCSEC between classes C1 and B3 makes the generation of test conditions for processor security testing a challenging task (i.e., class A1 requires that FTLS be produced for the user-visible hardware functions and thus these FTLS represent a source of test conditions). The generation of test data is somewhat less challenging because this activity is related to a specific coverage analysis method, namely boundary-value coverage, which implies that the test designer should produce test data for both positive and negative outcomes of any condition.

Lack of DTLS and of protection-model requirements for processors' hardware and firmware makes it important to identify various classes of security test conditions for processors that illustrate potential sources of test conditions. We partition these classes of test conditions into the following categories: (1) processor tests that help detect violations of TCB isolation and noncircumventability, (2) processor tests that help detect violations of policy, and (3) processor tests that help detect other generic flaws (e.g., integrity and denial of service flaws).

3.6.5.1 Test Conditions for Isolation and Noncircumventability Testing

- (1) There are tests which detect flaws in instructions that violate the separation of user and TCB (privileged) domain:

Included in this class are tests that detect flaws in bounds checking CPU and I/O processors, top- and bottom-of-the-stack frame checking, dangling references, etc. [4]. Tests within this class should include the checking of all addressing modes of the hardware/firmware. This includes single and multiple-level indirect addressing [3 and 4], and direct addressing with no operands (i.e., stack addressing), with a single operand and with multiple operands. Tests which demonstrate that all the TCB processor, memory, and I/O registers are inaccessible to users who execute nonprivileged instructions should also be included here.

This class also includes tests that detect instructions that do not perform or perform improper access privilege checks. An example of this is the lack of improper access privilege checking during multilevel indirections through memory by a single instruction. Proper page- or segment-presence bit checks as well as the proper invalidation of descriptors within caches during process switches should also be tested. All tests should ensure that all privilege checking is performed in all addressing modes. Tests which check whether a user can execute privileged instructions are also included here. Examples of such tests (and lack thereof) can be found in [3, 4, 22, and 23]

- (2) There are tests that detect flaws in instructions that violate the control of transfer between domains:

Included in this class are tests that detect flaws that allow anarchic entries to the TCB domain (i.e., transfers to TCB arbitrary entry points and at arbitrary times), modification and/or circumvention of entry points, and returns to the TCB which do not result from TCB calls. Tests show that the local address space of a domain or ring is switched properly upon domain entry or return (e.g., in a ring-based system, such as SCOMP, Intel 80286-80386, each ring stack segment is selected properly upon a ring crossing).

(3) There are tests that detect flaws in instructions that perform parameters validation checks:

Included in this class are tests that detect improper checks of descriptor privileges, descriptor length, or domain/ring of a descriptor (e.g., Verify Read (VERR), Verify Write (VERW), Adjust Requested Privilege Level (ARPL), Load Access Rights (LAR), Load Segment Length (LSL) in the Intel 80286-80386 architecture [24], Argument Addressing Mode (AAM) in Honeywell SCOMP, [22 and 23], etc.).

3.6.5.2 Text Conditions for Policy-Relevant Processor Instructions

Included in this class are tests that detect flaws that allow user-visible processor instructions to allocate/deallocate objects in memory containing residual parts of previous objects and tests that detect flaws that would allow user-visible instructions to transfer access privileges in a way that is inconsistent with the security policy (e.g., capability copying that would bypass copying restriction, etc.).

This class also includes tests that detect flaws that allow a user to execute nonprivileged instructions that circumvent the audit mechanism by resetting system clocks and by disabling system interrupts which record auditable events. (Note that the flaws that would allow users to access audit data in an unauthorized way are already included in Section 3.6.5.1 because audit data is part of the TCB.)

3.6.5.3 Tests Conditions for Generic-Security Flaws

Included in this class are tests that detect flaws in reporting protection violations during the execution of an instruction. For example, the raising of the wrong interrupt (trap) flag during a (properly) detected access privilege violation may lead to the interrupt (trap) handling routine to violate (unknowingly) the security policy. Insufficient interrupt/trap data left for interrupt/trap handling may similarly lead to induced violations of security policy by user domains.

Also included in this class are tests that detect flaws in hardware/firmware which appear only during the concurrent activity of several hardware components. For example, systems which use paged segments may allow concurrent access to different pages of the same segment both by I/O and CPU processors. The concurrent checking of segment privileges by the CPU and I/O processors should

be tested in this case (in addition to individual CPU and I/O tests for correct privilege checks [3]).

The *TCSEC* requirements in the area of security testing state that security testing and analysis must discover all flaws that would permit a user external to the TCB to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users. At the hardware/firmware level there are several classes of flaws (and corresponding tests) that could cause (detect) violations of this *TCSEC* requirement. The following classes of flaws are examples in this area. (Other examples of such classes may be found in future architectures due to the possible migration of operating system functions to hardware/firmware.)

- (1) There are tests that detect addressing flaws that place the processors in an "infinite loop" upon executing a single instruction:

Included in these flaws are those that appear in processors that allow multilevel indirect addressing by one instruction. For example, a user can create a self-referential chain of indirect addresses and then execute a single instruction that performs multilevel indirections using that chain. Inadequate checking mechanisms may cause the processor to enter an "infinite loop" that cannot be stopped by operating system checks. Lack of tests and adequate specifications in this area are also explained in [3].

- (2) There are tests that detect flaws in resource quota mechanisms:

Included in these flaws are those that occur due to insufficient checking in hardware/firmware instructions that allocate/deallocate objects in system memory. Examples of such flaws include those that allow user-visible instructions to allocate/deallocate objects in TCB space. Although no unauthorized access to TCB information takes place, TCB space may be exhausted rapidly. Therefore, instructions which allow users to circumvent or modify resource quotas (if any) placed by the operating system must be discovered by careful testing.

- (3) There are tests that detect flaws in the control of object deallocation:

Included in these flaws are those that enable a user to execute instructions that deallocate objects in different user or TCB domains in an authorized way.

Although such flaws may not cause unauthorized discovery/modification of information, they may result in denial of user communication.

3.6.6 Relationship Between Hardware/Firmware Security Testing and the TCSEC Requirements

In this section we review test condition and coverage analysis approaches for hardware/firmware testing. The security testing requirements for hardware/firmware are partitioned into three groups: (1) requirements for classes C1 and C2, (2) requirements for classes B1 to B3, and (3) requirements for class A1. For hardware/firmware security testing, the TCSEC does not allow the derivation of specific test-condition and coverage-analysis requirements for individual classes below class A1. The dearth of explicit general hardware/firmware requirements in the TCSEC rules out class-specific interpretation of hardware/firmware security testing requirements below class A1.

Security Classes C1 and C2

Test Conditions

For security classes C1 and C2, test conditions are generated from manual page descriptions of each processor instruction, and from the description of the protection mechanisms found in the processor's reference manuals. The test conditions generated for these classes include those which help establish the noncircumventability and the isolation (i.e., tamperproofness) of the TCB. These test conditions refer to the following processor-supported protection mechanisms:

- (1) Access authorization mechanisms for memory-bound checking, stack-bound checking, and access-privilege checking during direct or indirect addressing; and checking the user's inability to execute processor-privileged instructions and access processor-privileged registers from unprivileged states of the processor.
- (2) Mechanisms for authorized transfer of control to the TCB domain, including those checking the user's inability to transfer control to arbitrary entry points, and those checking the correct change of local address spaces (e.g., stack frames), etc.

(3) Mechanisms and instructions for parameter validation (if any).

Other test condition areas, which should be considered for testing the processor support of TCB noncircumventability and isolation, may be relevant for specific processors.

Test Coverage

The security testing guidelines of the *TCSEC* require that at least five specific tests be conducted for class C systems in an attempt to circumvent the security mechanisms of the system. This suggests that at least five test conditions should be included for each of the three test areas defined above. Each test condition should be covered by boundary-value coverage to test all positive and negative outcomes of each condition.

Security Classes B1 to B3

Test Conditions

For security classes B1 to B3, the test conditions for hardware/firmware security testing are generated using the same processor documentation as that for classes C1 and C2. Additional class-specific documentation is not required (e.g., DTLS is not required to include hardware/firmware TCB components that are visible at the TCB interface—unlike class A1).

The test conditions generated for classes B1 to B3 include all those that are generated for classes C1 and C2. In addition, new test conditions should be generated for the following:

- (1) Processor instructions that can affect security policy (e.g., instructions that can allocate/deallocate memory—if any), and instructions that allow users to transfer privileges between different protection domains, etc.
- (2) Generic security-relevant mechanisms (e.g., mechanisms for reporting protection violations correctly) and mechanisms that do not invalidate address-translation buffers correctly during process switches, etc.

(3) Mechanisms that control the deallocation of various processor-supported objects, and those that control the setting of resource quotas (if any), etc.

The only test conditions that are specific to the B1 to B3 security classes are those for hardware/firmware mechanisms, the malfunctions of which may allow a user to place the TCB in a state in which it is unable to respond to communication initiated by users.

Test Coverage

The security testing guidelines of the *TCSEC* require that at least fifteen specific tests should be conducted for class B systems in an attempt to circumvent the security mechanisms of the system. This suggests that at least fifteen test conditions should be included for each of the three test areas defined above (and for the three areas included in classes C1 through C2). Each test condition should be covered by boundary-value coverage to test all the positive and negative outcomes of each condition.

Security Class A1

The only difference between the hardware/firmware test requirements of classes B1 to B3 and those of class A1 are (1) the processor test conditions derived for classes B1 to B3 (which should also be included here) are augmented by the test conditions derived from DTLIS and FTLIS, and (2) the test coverage should include at least twenty-five test conditions for each test area (included in classes B1 to B3).

3.7 TEST PLAN EXAMPLES

In this section we present five test plan examples that have been used in security testing. An additional example is provided to illustrate the notion of cyclic test dependencies and suggest means for their removal. The first example contains a subset of the test plans for the **access** kernel primitive in Secure Xenix™. Here we explain the format of test conditions and of test data necessary for test conditions and focus on the notion of data flow analysis that might be presented in the coverage section of test plans.

The second example contains a subset of the test plans for the **open** kernel primitive of Secure Xenix™. Here we explain the use of the access-check

generated or TCB kernels to eliminate redundant tests without loss of coverage. In particular, we discuss the impact on test coverage of the dependency of the **open** kernel primitive on the **access** kernel primitive. For example, during the testing of the **access** primitive, the subpath starting at the "**obj_access**" check that includes the "**mand_access**" and "**discr_access**" functions is tested (Figure 2, page 24). Then the **open** primitive, which shares that subpath of the graph with the **access** primitive, need only be tested (1) for the access check that is not shared with the **access** primitive, and (2) to demonstrate that the data flow of **open** joins that of **access**. This can be done with only a few test conditions, thereby reducing the test obligation without loss of coverage.

The third example of this section explains a security test plan for the **read** kernel primitive of Secure Xenix™ systems. The specifications of the **read** primitive are access-check dependent on those of **open**. This means that a nonempty subset of the access checks necessary for **read** is done in **open** and, therefore, need not be tested again for **read**. To obtain the same coverage for **read** as that for **open**, or for **access**, one only needs to test (1) the existence of the access-check dependency of **read** on **open**, and (2) the remaining access checks specific to **read** (not performed in **open**). Since the testing of the access-check dependency requires only a few test conditions, the number of test conditions for **read** is reduced significantly without loss of coverage. The subset of the test plans required for **read** that are illustrated here is a subset of the test plans required for dependency verification. In contrast with the first two examples, which contain only mandatory access control (MAC) test conditions, this example includes some DAC test conditions.

The fourth example presents a subset of the test plans for the kernel and TCB isolation properties of Secure Xenix™. These test plans are derived from a set of kernel isolation and noncircumventability requirements for Secure Xenix™ and are important for at least three reasons. First, no formal model exists for these requirements for any system to date. This is true despite the fundamental and obvious importance of isolation and noncircumventability requirements in demonstrating the soundness of the reference monitor implementation in a secure system (at any and all security classes above B2). Second, test plans for these requirements cannot be generated from DTLS or FTLS of a secure system at any level (i.e., B2 to A1). This is because these requirements are not necessarily

specified in top-level specifications. This is true because isolation and noncircumventability properties include low-level dependencies on processor architecture details that do not correspond to the level of abstraction of TCB top-level specifications. Isolation and noncircumventability properties also cannot be verified formally using the current methodologies assumed by the tools sanctioned by the NCSC at this time (see the Appendix for the justification of unmapped kernel isolation code in the SCOMP specification-to-code correspondence example). Third, the kernel isolation and noncircumventability properties of a system depend to a large degree on the underlying processor architecture and on the support the architecture provides for kernel implementation. These test plan examples would therefore necessarily assume a given processor architecture. (An example of test or verification conditions for such processor mechanisms is provided by Millen in reference. [19])

In spite of the inherent architectural dependency of kernel isolation properties, we have selected a few examples of test plans that assume a very simple architecture and therefore can be generalized to other secure systems. The processor architecture, which is assumed by most implementations of the machine-dependent code of UnixTM systems, includes only the following: (1) a two-state processor (i.e., distinguished privileged mode versus unprivileged mode), (2) the ability to separate kernel address space (e.g., registers and memory) from user space within the same process, which could ensure that the kernel space cannot be read or written from user-space code, and (3) the ability to restrict transfers of control to specific entry points into the kernel. Other facilities are not assumed, such as special instructions that help the kernel and TCB primitive validate parameters and special gate mechanisms that help distinguish between privileged and nonprivileged kernel invocations.

Of necessity, the test plan examples for the above-mentioned kernel primitives and isolation examples are incomplete because it would be impractical to include complete test plans for these kernel primitives here.

The fifth example presents two test plans used for the processors of the Honeywell SCOMP system [22 and 23]. The first plan includes three test conditions for the ring crossing mechanism of the SCOMP processor and their associated test data and coverage analysis. The second plan presents a test condition for which

test programs cannot be generated in the normal mode of processor operation, illustrating the need for design analysis.

The last example of this section illustrates the notion of cyclic test dependencies that appear among test programs. It also shows how the use of the access-graph and access-check dependencies in defining test plans (especially coverage analysis) helps eliminate cyclic test dependencies.

3.7.1 Example of a Test Plan for "Access"

The **access** kernel primitive of Secure Xenix™ has two arguments, *path* and *amode*. The first argument represents the name of an object whose access privileges are checked by the primitive, whereas the second argument represents the privileges to be checked. The following types of objects can be referenced by the *path* names provided to **access**:

Files, Directories, Devices, Named Pipes, Xenix Semaphores, Xenix Shared Data Segments.

The following types of privileges and combinations thereof are checked by **access**:

Read, Write, Execute (and object's existence).

3.7.1.1 Test Conditions for Mandatory Access Control of "Access"

The following test conditions are derived from the interpretation of the mandatory access control model [12].

- (1) Whenever the type of object named by *path* is one of the set {File, Directory, Device}, then the **access** call succeeds when executed by a process that wants to check the existence of *amode* = Read, Execute privileges, or the existence of the object, if process clearance dominates the object classification; otherwise, the **access** call fails.
- (2) Whenever the type of object named by *path* is one of the set {Named Pipe, Xenix Semaphore, Xenix Shared Data Segment}, then the **access** call succeeds when executed by a process that wants to check the existence of

amode = Read privilege, or the existence of the object, if process clearance equals the object classification; otherwise, the **access** call fails.

(3) Whenever the type of object named by *path* is one of the set {File, Directory, Device, Named Pipe, Xenix Semaphore, Xenix Shared Data Segment}, then the **access** call succeeds when executed by a process that wants to check the existence of *amode* = Write privilege, or the existence of the object, if process clearance equals the object classification; otherwise, the **access** call fails.

3.7.1.2 Test Data for MAC Tests

Environment Initialization

A subset of all clearances and category sets supported in the system is defined in such a way as to allow all relationships between security levels to be tested (e.g., level dominance, incompatibility, equality). For example, the chosen levels are UNCLASSIFIED/Null/, UNCLASSIFIED/B/, CONFIDENTIAL/A, B/, SECRET/Null/, SECRET/All/, TOP SECRET/A/, and TOP SECRET/A, B/. The security profile of the test operator is defined to allow the test operator to login at all of the above levels.

A subset of all directory hierarchies supported in the system is defined in such a way as to allow all relationships between objects of the hierarchy to be tested (e.g., child and parent directories—see the discussion in Section 3.5.2.2). Three directories, denoted as directory 1, 3, and 6, are created from the “home” directory at levels UNCLASSIFIED/A, B/, SECRET/A/, and TOP SECRET/A, B/. Two directories, denoted as directory 4 and 5, are created from the SECRET/A/ directory 3 at levels SECRET /All/ and TOP SECRET /A/. An additional CONFIDENTIAL /A, B/ directory, denoted as directory 2, is created from the UNCLASSIFIED/A, B/ directory.

The test operator logs in at each of the above security levels and creates a file in each directory. The discretionary access privileges are set on every file and directory to allow all discretionary checks performed by the TCB to succeed. The directory hierarchy is thus created, and the definitions of the security levels for each file and directory is shown in Figure 4.

Test Documents

The test operator logs in at each of the above security levels and invokes the **access** call with the following parameters:

path: Every file pathname defined in the hierarchy.

amode: All access privileges individually and in combination.

Outcomes

Tables 1 and 2 show the expected outcomes of **access** indicating the correct implementation of mandatory access checks. Note that in Tables 1 and 2 "Fail 1" errors should provide no information about the nature of the failure. This is the case because these failures are returned whenever the invoker of access is at a lower level than that of the object being accessed. In particular, "Fail 1" should not indicate the existence or nonexistence of files at levels that are higher than, or incompatible with, the login level. Discovery of an object's existence or nonexistence at a higher level than that of the accessor's would provide a covert storage channel. In contrast, "Fail 2" errors allow the invoker of **access** to discover the existence of the

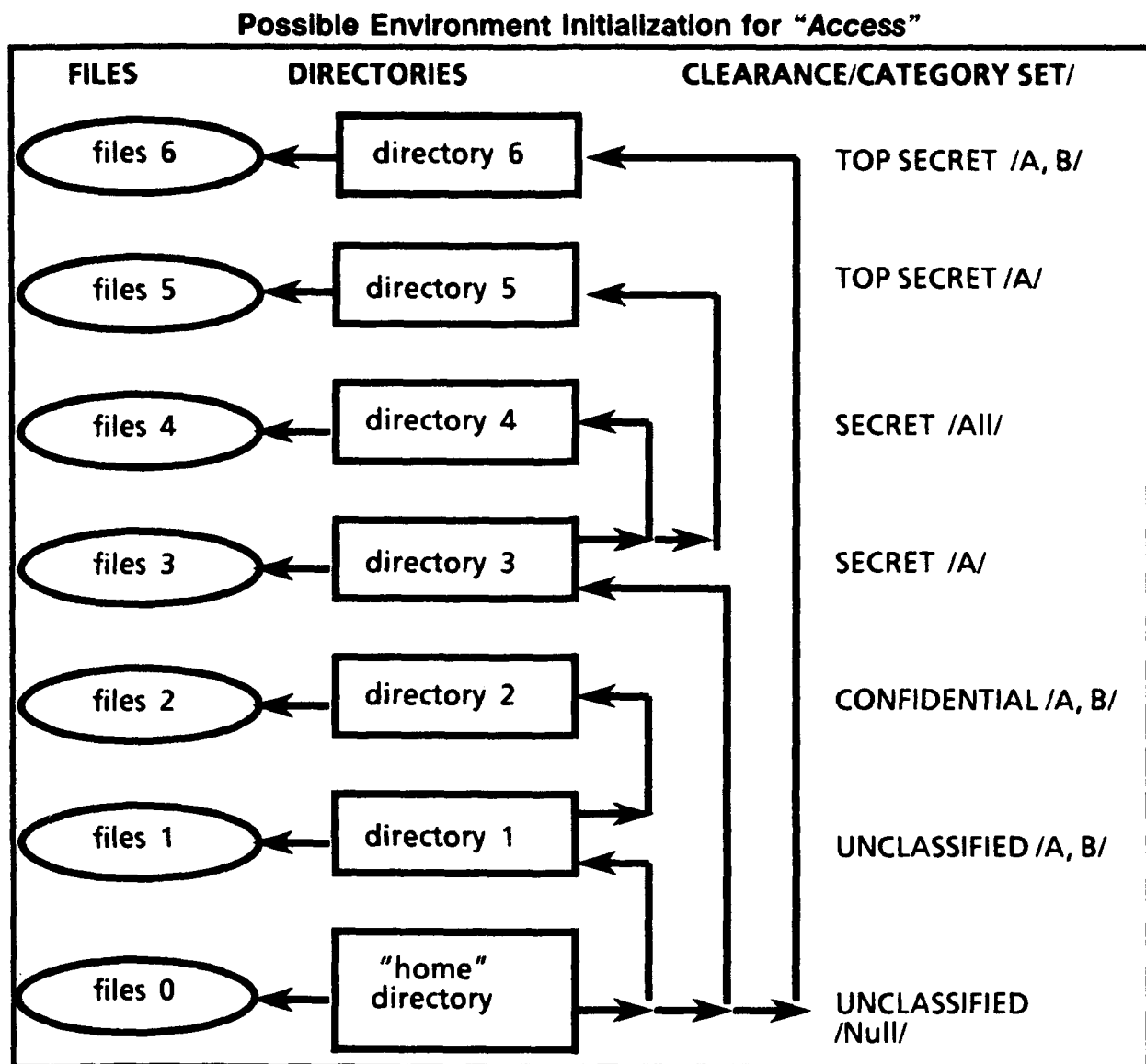
file, because his security level dominates that of the file. No covert channel provided by the object's existence is possible in this case.

3.7.1.3 Coverage Analysis

Model-Dependent MAC Coverage

The test conditions provided above cover all MAC checks for the **access** primitive. The test data of this plan, however, cover the test conditions only partially. For example, condition (2) is not covered at all because the object types included in the test data are only files. Conditions (1) and (3) are only partially covered for the same reason. Environment reinitialization is necessary to allow the testing of **access** with all other types of objects.

The above test data also provide partial coverage because they do not include the hierarchies shown in Figures 3a and 3b, page 35. Re-execution of the above tests with the hierarchies shown in Figures 3a and 3b would guarantee sufficient

**Figure 4**

coverage of the MAC model hierarchy. The test parameters and outcomes will generally differ if the additional tests suggested here are carried out.

Call-Specific MAC Coverage

Let us consider the coverage of individual arc paths and of combinations of arc paths as required by data flow coverage of the **access** primitive. The question as to whether the graph of the **access** primitive shown in Figure 2, page 24, is covered adequately or redundantly by the above test conditions and data arises naturally. The following three cases of coverage illustrate primitive-specific coverage analysis for test condition (1) and test data of Figure 4, page 55, and Table 1, page 57.

Case 1. A Single Arc Path

The test operator logs in at level UNCLASSIFIED/A, B/ and invokes **access** with **read** as a parameter on the file/home/directory1/directory2/file2 at level CONFIDENTIAL/A, B/. As shown in Figure 4 and Table 1, **access** is at the level of the invoking program (i.e., UNCLASSIFIED/A, B/) and, therefore, the call to it will fail.

**Outcomes of "Access" with Read, Execute, Read-Execute Options
and File-Existence Checking**

access with test program at level:	File at Un/Null/	File at Un/A, B/	File at Confid/ A, B/	File at Secret /A/	File at Secret/ All	File at TS/A/	File at TS/A, B/
Un/Null/	Succ	Fail 1	Fail 1	Fail 1	Fail 1	Fail 1	Fail 1
Un/A, B/	Succ	Succ	Fail 1	Fail 1	Fail 1	Fail 1	Fail 1
Confid/A,B/	Succ	Succ	Succ	Fail 1	Fail 1	Fail 1	Fail 1
Secret/A/	Succ	Fail 1	Fail 1	Succ	Fail 1	Fail 1	Fail 1
Secret/All/	Succ	Succ	Succ	Succ	Succ	Fail 1	Fail 1
TS/A/	Succ	Fail 1	Fail 1	Succ	Fail 1	Succ	Fail 1
TS/A, B/	Succ	Succ	Succ	Succ	Fail 1	Succ	Succ

Table 1

Outcomes of "Access" with Write, Read-Write, Read-Write-Execute Options

access with test program at level:	File at Un/Null/	File at Un/A, B/	File at Confid/A, B/	File at Secret/A/	File at Secret/All	File at TS/A/	File at TS/A, B/
Un/Null/	Succ	Fail 1	Fail 1	Fail 1	Fail 1	Fail 1	Fail 1
Un/A, B/	Fail 2	Succ	Fail 1	Fail 1	Fail 1	Fail 1	Fail 1
Confid/A,B/	Fail 2	Fail 2	Succ	Fail 1	Fail 1	Fail 1	Fail 1
Secret/A/	Fail 2	Fail 1	Fail 1	Succ	Fail 1	Fail 1	Fail 1
Secret/All/	Fail 2	Fail 2	Fail 2	Fail 2	Succ	Fail 1	Fail 1
TS/A/	Fail 2	Fail 1	Fail 1	Fail 2	Fail 1	Succ	Fail 1
TS/A, B/	Fail 2	Fail 2	Fail 2	Fail 2	Fail 1	Fail 2	Succ

Table 2

This test provides a single arc path coverage, namely that of arc path **access** -> "namei" -> "obj__access" -> "mand__access," shown in Figure 2. Here "mand__access" returns "failure" when it tries to resolve the file path name. Note that the file path name component "file2" cannot be read from directory "directory\2" because the mandatory check fails. Note that mandatory checks on the level of the file itself are also not performed here. The mandatory check failure is caused earlier by path name resolution and returned to "namei."

Case 2. A Combination of Arc Paths

The test program logs in at level TOP SECRET/A/ and invokes **access** with Read as a parameter on the file /home/directory3/file3 at level SECRET/A/. **Access** is at the level TOP SECRET/A/ (Figure 4 and Table 1), therefore, the call to it will succeed.

This test provides multiple arc path coverage. The first arc path is the same as in Case 1 above. The "mand__access" check passes, however, and control is returned all the way up to access; see Figure 2. The second arc path is "access" -> "obj__access" -> "mand__access." The mandatory check in "mand__access" is performed directly on the file and not on its parent directory as in Case 1. This

check succeeds. The success result returns to "obj__access" which initiates a third arc path traversal to "discr__access." The discretionary check passes (as set up in the environment definition) and success is returned to "obj__access" and **access**.

Case 3. A Different Combination of Arc Paths

The test program logs in at level SECRET/A/ and invokes **access** with Read as a parameter on the directory /home/directory3/directory5 at level TOP SECRET/A/. As shown in Figure 4 and Table 1, **access** is at the level SECRET/A/. The call to it will fail.

Although this test appears to provide the same coverage as that of Case 1, in fact it does not. The first arc path is the same as that in Case 1 above, except that the "mand__access" check on the path name of the target object (which terminates with name "directory5" in directory/home/directory3) succeeds and control is returned all the way up to **access** (see Figure 2, page 24). The second arc path is then "access" -> "obj__access" -> "mand__access." The check in "mand__access" is performed directly on the directory /home/directory3/directory5 and, unlike the check in Case 2, it fails. This "failure" is returned to "obj__access" which reports it to **access**. Coverage analysis based on a specific model interpretation in a given TCB primitive would require that the Case 1 test be repeated with a directory replacing the file "file2." However, this new Case 1 test would become indistinguishable from that of Case 3 in coverage analysis based on abstract models, and thus Case 3 would not necessarily be tested.

3.7.2 Example of a Test Plan for "Open"

The kernel primitive **open** has as arguments a *path*, *oflag*, and *mode*. The only relevant object types named by *path* for **open** are the following:

Files, Directories, Devices, and Named Pipes.

The *oflag* parameter denotes different access privileges and combinations thereof. It also contains other flags such as "o__excl," "o__creat," and "o__trunc" that specify object locking, default creation, or truncation conditions. The *mode* argument of the **open** primitive is relevant only when the object does not exist and the "o__creat" flag is used.

3.7.2.1 Test Conditions for "Open"

Test Condition for Access-Graph Dependency

Verify that the **open** kernel primitive shares the **access** primitive subgraph that includes the object `__access` checks.

Examples of Test Conditions Specific to "Open"

- (1) Verify that if the object specified by the *path* argument does not exist, the object is created with the access privileges specified by the *mode* argument whenever the "o__creat" flag is on, with the owner's user ID and the specified group ID, and with the invoker process' security level.
- (2) Verify that if the object specified by the *path* argument exists, the **open** kernel primitive succeeds whenever the requested privilege specified by the "o__flags" is granted to the person with access. Verify that, in this case, the *mode* argument also has no effect on the existing privileges of the object.
- (3) Verify that the **open** kernel primitive always fails when it is invoked:
 - With the "write" access privilege for a Directory.
 - On a nonexistent device.
 - On Xenix Semaphores and Xenix Shared Data Segments.

3.7.2.2 Test Data for the Access-Graph Dependency Condition

Environment Initialization Parameters

A subset of all clearances and category sets defined in the tests of **access** is chosen in such a way as to allow all relationships between security levels to be tested (e.g., level dominance, incompatibility, equality). The chosen levels are UNCLASSIFIED/Null/, SECRET/All/, and TOP SECRET/A/. The subset of the directory hierarchy defined in the tests of **access** that contains the objects at the above chosen levels is selected for this test. The definition of the security levels and discretionary access privileges and the creation and initialization of the object hierarchy are performed in a similar way to that used in the test of **access**. The

security profile of the test operator is defined to allow him to login at all of the above levels.

The test operator logs in at each of the chosen security levels and invokes the **open** primitive with the following parameters:

path: path names of the three files defined in the hierarchy.

o_flags: *o_read*, *o_write*, individually, and in the following combinations:

o_read| *o_excl*, *o_read*|*o_trunc*, *o_read*|*o_excl*|*o_trunc*,

o_write| *o_excl*, *o_write*|*o_trunc*, *o_write*|*o_excl*|*o_trunc*,

For example, the test operator will use the following login, security level, files, and “*o_flag*” parameters:

Case 1

The test program logs in at level SECRET/All/ and invokes **open** on file /home/directory3/directory5/file5 at level TOP SECRET/A/ with *o_read_only* as the call option. **Open** is at the level of the invoking program as shown in Table 3, therefore, the call fails.

Outcomes for “Open” with Read Option

<i>open</i> with test program at level:	File at Uncl/Null/	File at Secret/All/	File at Top Sec/A/
Uncl/Null/	Succ	Fail 1	Fail 1
Secret/All/	Succ	Succ	Fail 1
Top Sec/A/	Succ	Fail 1	Succ

Table 3

Case 2

The test program logs in at level SECRET/All/ and invokes **open** on file /home/file0 at level UNCLASSIFIED/Null/ with *o_read_only* as the call option.

Open is at the level of the test program as shown in Table 3, therefore, the call succeeds.

(Note that the above failure and success of the **open** invocations occur for the same reasons as those explained in Cases 1 and 2 of the Coverage Analysis area of the **access** test plan in Section 3.2.4.1.)

Case 3

The test program logs in at level **SECRET/All/** and invokes **open** on file **/home/directory3/directory4/file4** at level **SECRET/All/** with "**o_read_only**" as the call option. **Open** is at the level of the test program as shown in Table 4, therefore, the call succeeds.

Tables 3 and 4 show the expected outcome of **open** indicating the consistency of these outcomes with those of **access**. Note that, as in the outcomes of **access**, "Fail 1" errors in Tables 3 and 4 provide no information about the nature of the failure, which otherwise might indicate the existence or nonexistence of files at levels that are higher than, or incompatible with, the login level. In contrast, "Fail 2" errors allow the invoker of **open** to discover the existence of the file, because the user security level dominates that of the file.

Outcomes for "Open" with Write Option

<i>open</i> with test program at level:	File at Uncl/Null/	File at Secret/All/	File at Top Sec/A/
Uncl/Null/	Succ	Fail 1	Fail 1
Secret/All/	Fail 2	Succ	Fail 1
Top Sec/A/	Fail 2	Fail 1	Succ

Table 4

3.7.2.3 Coverage Analysis

Model-Dependent MAC Coverage

The testing of the access graph dependency of **open** on **access** provides the same model-dependent MAC coverage for **open** as that provided for **access**. That is, after **access** is fully tested using data flow coverage, the same coverage is obtained for **open**. Access-graph dependency testing confirms that the access subgraph shared by the two primitives, which includes the "obj_access" function, enforces the MAC policy. Since all object types relevant to **open** are included among those of **access**, and since all access modes of **open** are included among those of **access** (the "exclusive" and "truncation" modes introducing no additional modes independent of read and write), the only additional model dependent MAC tests necessary for **open** are those which confirm the access-graph dependency of **open** on **access** for the remaining types of objects (i.e., Directories, Devices, and Named Pipes).

Call-Specific MAC Coverage

Additional primitive-specific test data are necessary to demonstrate that MAC policy is discovered by the test plans. Test data, for example, must be provided for test conditions (2) and (3) above.

3.7.3 Examples of a Test Plan for "Read"

The kernel primitive **read** used the file descriptor *filides* to identify the target object of the read action. The file descriptor is obtained from the kernel primitives **open**, **creat**, **dup**, **fcntl**, and **pipe**. Since the primitives **dup** and **pipe** are not access-control-relevant, and since **fcntl** is tested elsewhere, the only primitives and object types relevant to **read** are the following:

open, **creat** Files, Directories, Devices, and Named Pipes.

The **read** primitive uses *filides* as one of its parameters, which is obtained from either **open** or **creat**. Thus, **read** can be called only after either of these two calls have been performed successfully. This establishes the access-check dependency condition—the only test condition that will be included in the test plan.

3.7.3.1 Test Conditions for "Read"

For each object type in the set {Files, Directories, Devices, Named Pipes}, verify the following:

- (1) That **read** fails, if neither **open** nor **creat** call has been performed before the **read** call.
- (2) That **read** fails, if neither **open** nor **creat** call returned "success" before the **read** call.
- (3) That **read** succeeds whenever an **open** call including the read privilege has been successfully performed (**creat** has no read option).

3.7.3.2 Test Data for the Access-Check Dependency Condition

Environment Initialization for Condition (1)

The test operator logs in as *uid1.gid1* at a given security level, such as UNCLASSIFIED/Null/, and attempts to read a file without calling **open** or **creat** first. An account for the test operator must exist. Note that the initialization of the discretionary privileges is irrelevant only for the first condition. Figure 5a illustrates the initialized environment.

Data for Uninitialized File Descriptors			
Discretionary Access	File Name	Directory Name	Security Level
		<div style="border: 1px solid black; padding: 2px; display: inline-block;"> "home" UID1. GID1 </div>	Unclassified/NULL

Figure 5a Parameters for Condition (1) Tests

Test Parameters for Condition (1)

After environment initialization, or program calls **read** with the parameters, *files* = 3, . . . , 20. Note that descriptors 0, 1, 2 are already opened for the standard input, output, and error files and therefore cannot be used here. Note that *files* 20

Outcomes for Uninitialized File Descriptors

Test Operator Identifier	File Descriptor Number	Outcomes of <i>read</i>
UID1. GID 1	3	F
UID1. GID 1	4	F
UID1. GID 1	5	F
.	.	.
.	.	.
.	.	.
UID1. GID 1	19	F
UID11. GID1	20	F'

Table 5a

is also an invalid file descriptor but is included here to test that the **read** call fails when the file descriptor is out of range (i.e., a read specific test).

Outcomes for Condition (1) Tests

Table 5a shows the expected outcomes of the condition (1) test. "S"/"F" denotes a success/failure result.

Environment Initialization for Condition (2)

A file (denoted as file2) is created in the test operator's "home" directory with "read-only" discretionary access privilege initialized for the test operator. Testing consists of a logon as *uid1.gid1* at security level UNCLASSIFIED/NULL/ followed by two attempts to call **open** and **creat** in such a way that these calls fail. These calls will be followed by two attempts to call **read** with the *files* presumed to be returned by the calls to **open** and **creat**. Figure 5b summarizes the initialization needed for the required test.

Parameters for Condition (2) Tests

After environment initialization, the test operator or program performs the following actions using the underlined parameters:

Data When Either "Open" or "Creat" is Called and Fails

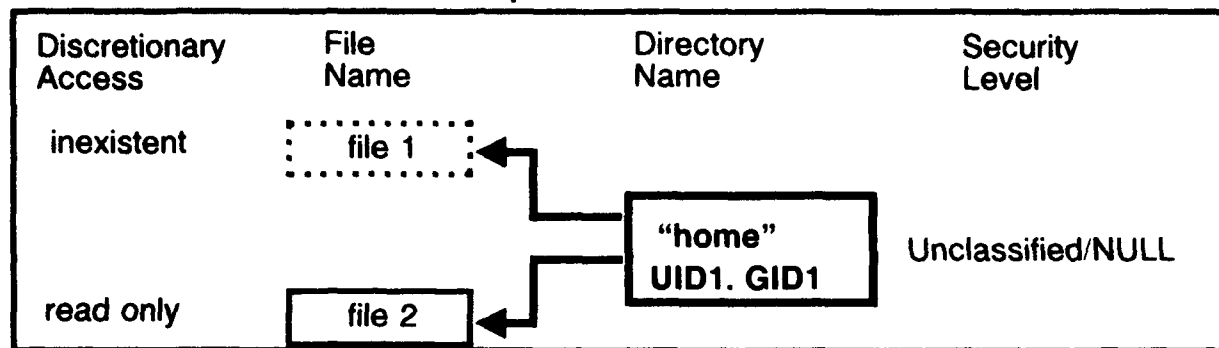


Figure 5b

- Open a nonexisting file (file1) using the **open** call with "o_read_only" flag (**open** fails because file1 does not exist). Then call **read** with the *filides* returned by the **open** call.
- Create a file (denoted as file2) with any arbitrary mode using the **creat** call (**creat** fails because file2 already exists and it has "read-only" privileges for the test operator). Then the test operator or program calls **read** with the *filides* returned by the **creat** call.

Outcomes for Condition (2) Tests

Testing demonstrates that **read** will fail in both cases because both **open** and **creat** returned "failure" earlier. Table 5b shows the expected outcomes.

Outcomes when either "Open" or "Creat" is Called and Fails

Logon Identifier	File Name	Opened by	Open option	File Descriptor	Outcomes of read
UID1. GID 1	file 1	<i>open ()</i>	<i>o_read_only</i>	filides 1	Fail
UID1. GID 1	file 2	<i>creat()</i>	<i>any</i>	filides 2	Fail

Table 5b

Environment Initialization for Condition (3)

Two files (denoted as file1 and file2) are created in the user's "home" directory with "read-only" and "write-only" discretionary access privileges respectively, defined for the test operator. The file security levels are defined in such a way that

all mandatory access checks succeed on both files. Testing requires a logon as *uid1.gid1* at the security level UNCLASSIFIED/NULL/ followed by two calls to **open** and one to **creat**. Figure 5c describes the data needed for the required tests.

Test Parameters for Condition (3) Test

After environment initialization, the test operator or program performs the following actions with the following parameters:

- Open file1 using the **open** call with "o_read_only" flag (**open** succeeds and returns a valid *fildev*), then call **read** with the *fildev* returned by the **open** call.
- Open file2 using the **open** call with "o_read_only" flag (**open** succeeds and returns a valid *fildev*), then call **read** with the *fildev* returned by the **open** call.
- Create a file (denoted as file3) in the test operator's "home" directory with all the discretionary access modes permitted using the **creat** call (**creat** succeeds and returns a valid *fildev*), then call **read** with the *fildev* returned by the **creat** call.

Data When "Open" or "Creat" Succeeds

Discretionary Access	File Name	Directory Name	Security Level
read only	file 1	"home" UID1.GID1	Unclassified/NULL
write only	file 1		
R/W/E	file 3		
.....		
.....	shows inexistent files		

Figure 5c Parameters for Condition (3) Tests

Outcomes for Condition (3) Tests

Testing demonstrates that **read** succeeds when the file descriptors from **creat** and **open** include the **read** privilege; otherwise, **read** fails. Since **open** succeeds before calling the **read** kernel primitive, **read** also succeeds only when an **open** call

was performed with the **read** option flag; otherwise, **read** fails. Although **creat** succeeds, **read** still fails because **creat** always opens an object for **write** only, and thus **read** actions are not permitted. Table 5c shows the expected outcomes.

Outcomes When "Open" and "Creat" Succeed

Logon Identifier	File Name	Opened by	Open Descriptor	File Descriptor	Outcomes of read
UID1. GID 1	file 1	<i>open</i> ()	<i>o_read_only</i>	filides 1	Succ
UID1. GID 1	file 2	<i>open</i> ()	<i>o_write_only</i>	filides 2	Fail
UID1. GID 1	file 3	<i>creat</i> ()	All	filides 3	Fail

Table 5c

3.7.3.3 Coverage Analysis

Model-Dependent Coverage

The testing of the access-check dependency of **read** on **open** and **creat** provides the same model-dependent coverage for **read** as that provided for **open** and **creat**. (Only a subset of this coverage is explained in Section 3.7.2.3.) The testing of the access-check dependency confirms that the **read** primitive cannot succeed unless the access checks that it requires have already been done in **open** and **creat**. Since all object types relevant to **read** are included among those of **open**, and since the read access privilege is covered in **open**, the only additional model-dependent tests necessary for **read** are those performed by hardware (e.g., read authorization checks) and those that confirm the access-check dependency of **read** on **open** and **creat** for the remaining types of objects (i.e., Directories, Devices, Named Pipes).

Call-Specific Coverage

Additional primitive-specific tests may be necessary for **read** depending upon its implementation. For example, if the object-limit check performed by **read** is in any way different from those of other primitives, it would need to be tested separately. Other conditions referring to object locking may also be included in these primitive-specific tests.

3.7.4 Examples of Kernel Isolation Test Plans

The test conditions presented in the following example refer to the transfer of control from a user-level program to the kernel of Secure Xenix™. Such transfers should take place only to entry points determined by the system design.

The kernel code and data segments of Secure Xenix™ are placed in ring (i.e., privilege level 0) whereas user-level code and data segments are placed in ring 3. User-level programs, therefore, cannot access kernel programs and data directly without transferring control to kernel programs first (this property is assured by the processor security testing). The transfer of control from user-level programs to the kernel can only take place in the following three ways:

- Through calls to a gate in the Global Descriptor Table (GDT), which is located in kernel address space, via segment selector number 144.
- Through software interrupts controlled by gate as in the Interrupt Descriptor Table (IDT) located in kernel address space.
- Through traps, which occur as the result of exceptional conditions and which either cause the kernel to terminate the user process execution immediately or cause the kernel to receive signals which eventually terminate the user process execution.

The test plans shown below illustrate that the above cases of transfer of control are the only ways to transfer control to the kernel.

3.7.4.1 Test Conditions

(1) Call Gate. This tests that application programs cannot successfully access any GDT descriptor except that provided by the segment selector number 144.

(2) Software Interrupts. These test that whenever an application program uses the interrupt-generating instructions "INT n" and "INTO," with $n \neq 3, F0-FB$, a general protection trap will occur. For $n=3$, the calling process will receive a SIGTRAP signal (a trap signal) and for $N=F0-FB$, will cause the instructions following "INT n" to be interpreted as 80287 instructions. (Note: The 80287 is the arithmetic co-processor.)

(3) Traps. These verify that the occurrence of traps will only affect the trap-generating process. (Note: This condition cannot be tested by user-level test programs because the traps cause the termination of the process running the test program. This condition can therefore only be verified by review of the source code files containing machine dependent code, i.e., mdep/trap.c and mdep/machdep.asm in Secure Xenix™.)

(4) Call validity. This tests that whenever a user-level program invokes the kernel with an invalid kernel number, the call will fail.

3.7.4.2 Test Data

Environment Initialization for Conditions (1), (2) and (4)

Compile the test program using "cc -M2es -i" flags. These compilation flags refer to the small memory model for C programs, with far pointers allowed and with separate instructions and data segments. The code segment selector number for the program code will be 0x3F. The data segment selector number will be 0x47. For each test condition, the program forks a child process to perform each test as described below.

Test Parameters

The following sequences describe the steps of the test programs and the test parameters for each condition:

- (1) Loop with an index value from 0x8 to 0x190 incrementing the index value by 8. Access a memory location whose segment selector number is provided by the index value.
- (2) Loop with an index value from 0 to 0xEF incrementing the index value by 1. Execute instruction "INT n" in the loop, where the value of n is provided by the index value.
- (3) No test condition or parameters are necessary (namely, the Note of test condition (3) above).
- (4) Invoke the kernel gate with the following INVALID kernel call numbers:

- 0x41 (outside of "sysent," the main kernel call table).
- 0x2228 (outside of "cxenix," the Xenix™ system call table).
- 0x1740 (outside of "csecurity," the security system call table).

Then invoke the kernel with VALID kernel call numbers representing a user-visible kernel call and a privileged kernel call.

Outcomes for the Test Conditions (1), (2), and (4) Above

(1) The process running the test program will receive a SIGSEGV signal (a segment violation signal) for each access call except when the gate selector number is 0x90.

(2) The following is true for the process running the test for the software interrupts:

- Will receive a SIGSEGV signal for each index value except for $n = 3$.
- Will receive a SIGTRAP signal when $n = 3$.
- Will not receive any signal when $n = 0xF0 - 0xFB$, because these index values represent valid entries for the 80287 arithmetic co-processor.

(3) No outcomes, since no tests are performed.

(4) Error EINVAL (i.e., invalid entry) will be received for all INVALID kernel call numbers (i.e., numbers outside the entry ranges of the main kernel call table, the Xenix™ system call table, and the security system call table). No error will be received for the kernel call using the valid kernel call number (i.e., for any number within the table entry range). An error will be received for the invocation of any privileged kernel call (primitive).

3.7.4.3 Coverage Analysis

The coverage of the above test conditions is based on boundary-value analysis. The test data place each test program above (i.e., successful outcome) and below (i.e., unsuccessful outcome) each boundary condition. The test data and outcomes represent the following degrees of condition coverage:

- (1) All kernel-call gate selection cases are covered.

- (2) All software interrupt selection cases are covered.
- (3) Not applicable (namely, the Note of condition (3) above).
- (4) All the boundary conditions are covered. For complete coverage of each boundary condition, all privileged kernel calls should be invoked, and all relevant out-of-range call numbers should be tested. (Such tests are unnecessary because the range tests in kernel code use the table ranges as defined constants.)

3.7.5 Examples of Reduction of Cyclic Test Dependencies

Consider the structure of typical test programs such as those for the **open**, **read**, **write**, **close**, and **fcntl** TCB primitive of UnixTM to illustrate cyclic test dependencies and their removal. The test program for each TCB primitive is denoted by t_n where n is the first character of the function name.

The test program for **open**, t_o , opens an object, for instance a file, writes on it a predetermined string of characters and closes the file. Then, it opens the file again and reads the file contents to ensure that the correct file has been opened. Thus, t_o must invoke the TCB primitive **write**, **close**, **read** in addition to **open**. The same sequence of TCB primitives is used for t_r and t_w to confirm that the **read** and **write** TCB primitives use the correct file. Note that, even if a single file is created in the test environment, the file system contains other system files that may be inadvertently read by the kernel. Thus, the use of a predetermined string of characters is still necessary to identify the file being written or read.

The test program for **close**, t_c , has a similar structure to that of t_o . After t_c opens an object (for instance a file) and writes on it a predetermined string of characters, t_c reads the string and closes the file. Then t_c opens the file again and reads the string. t_c must read the predetermined string of characters both before closing the file and after reopening the file to ensure that the correct file was closed. Even though **close** is a security-model-irrelevant TCB primitive, it must still be tested here since the test programs t_o , t_r , and t_w rely on it.

The TCB primitives **open**, **read**, and **write** are among the first to be tested because most other test programs depend on them. If no access-graph or access-check dependencies are used, t_o , t_r , t_w and t_c depend on each other as shown in Figure 6a. Note that the **fcntl** TCB primitive could have been used instead of **read**

in t_c . However, this would not have decreased the total number of cyclic test dependencies because the removed cyclic dependency between t_c and t_r would have to be replaced by the cyclic dependency between t_c and t_{ctl} .

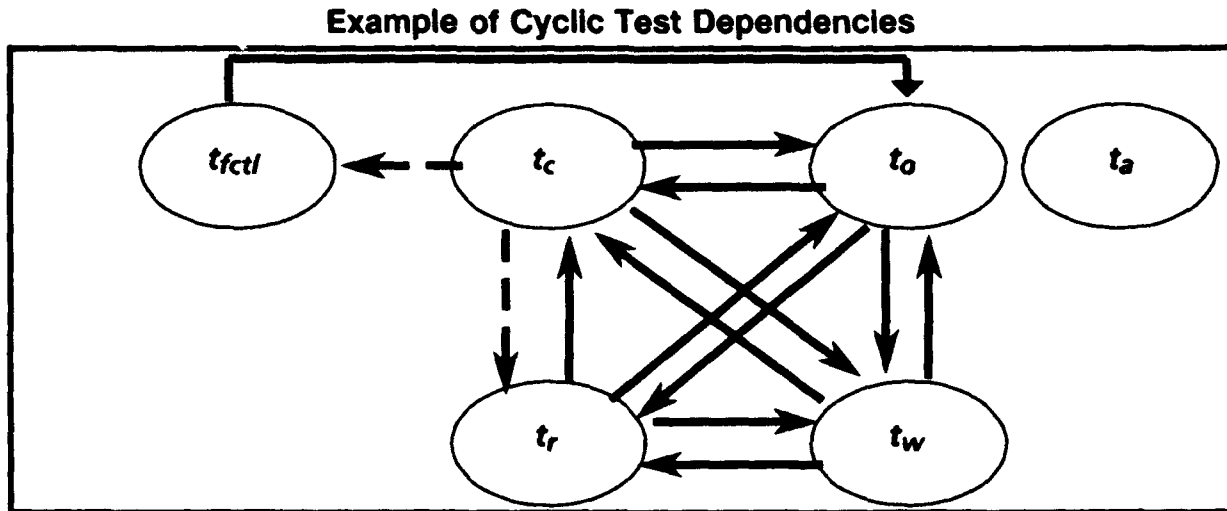


Figure 6a

The structure of the above test programs is not unique. Some of the cyclic test dependencies presented above, therefore, may not appear in other test programs. Other cyclic test dependencies similar to the ones shown above, however, will still appear. The reason for this is that kernel isolation and noncircumventability cause a test program for some TCB primitives to rely on other TCB primitives, and vice versa, whenever the TCB primitives are tested monolithically.

The use of the access-check graph for testing **open** eliminates the need to invoke the TCB primitives **read**, **write**, and **close** in t_o , and makes t_o dependent only on t_a , the test program for **access**. For example, since the access-check graph shows that both **open** and **access** use the same function for the resolution file names [i.e., `namei()`], the **read** and **write** primitives are unnecessary for file identification because the file name resolution has already been tested by t_a . Figure 6b shows the remaining cyclic dependencies between the test programs after all cyclic dependencies of t_o are removed.

The use of the access-check dependencies between TCB primitives also helps remove cyclic dependencies between test programs. For example, in tests for **read** and **write**, only the **open** TCB primitive needs to be used to test the existence of

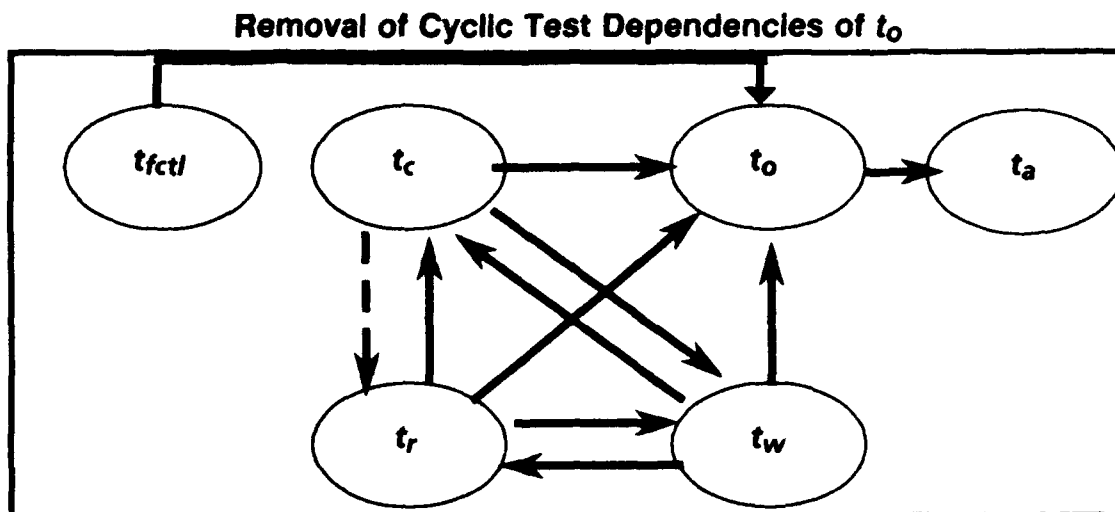


Figure 6b

the dependency in addition to those necessary to set up the test environment (e.g., **creat**). Figure 6c shows the remaining dependencies between the test programs t_a , t_r , t_w , t_o , and t_{fctl} . Note that since test programs t_o , t_r , and t_w do not use the TCB primitive **close**, and since **close** is security-model-irrelevant, the testing of **close** need not be performed at all. Note that the remaining test dependencies are generally not always identical to the ones shown in Figure 1, page 23. More dependencies than those shown in Figure 1 will remain after the new test method is applied.

The example of the remaining test dependencies shown in Figure 6c does not imply that the test program for **access**, t_a , invokes only **access**. T_a must also invoke TCB primitives needed to set up the test environment; therefore, it depends on the test programs for **creat** (t_{cr}), **ACL_control**, and on those of trusted processes **login** and **mkdir**. Also, t_{cr} depends on the primitive **access** because primitive **creat** shares a subgraph with **access**, Figure 2, page 24; therefore, the test program for **creat** (t_{cr}) depends on the test program for **access**. A cyclic test dependency therefore exists between t_a and t_{cr} (not shown in Figure 6c).

To eliminate all such cyclic test dependencies, a small routine with limited functionality, which is verified independently, could be added to the kernel to read out all the created test environments. The actions of the test programs that set up the test environments could then be verified independently. Judicious use of such a limited function **read** routine and of the new test method could lead to the

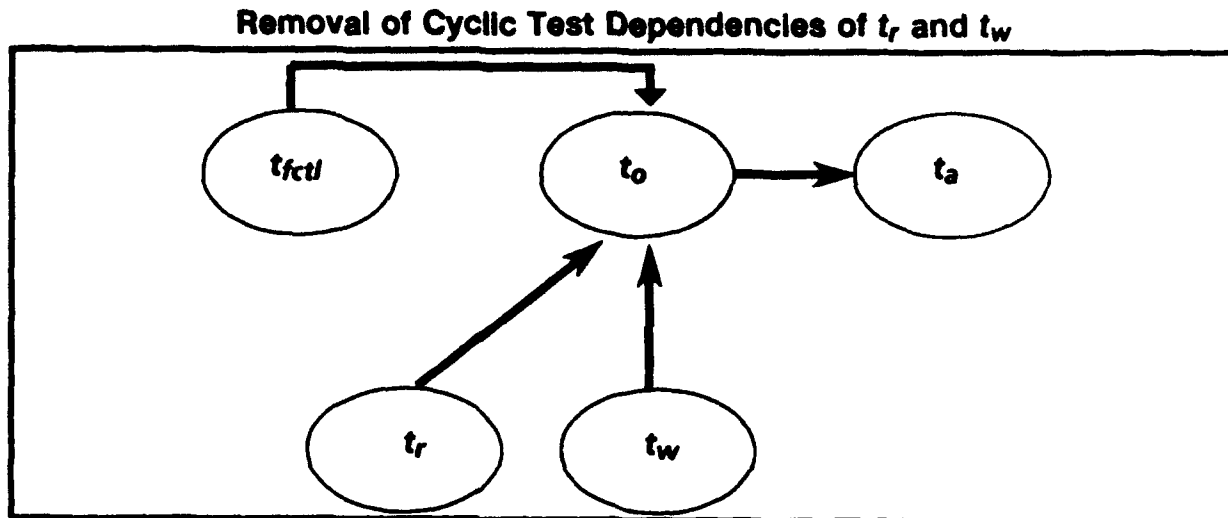


Figure 6c

elimination of all cyclic test dependencies. The addition of such a routine to the TCB, which could be done only in maintenance mode, would defeat our goal of test repeatability.

3.7.6 Example of Test Plans for Hardware/Firmware Security Testing

In the SCOMP documentation of processor security testing, the test conditions are identified by the “verify” clauses. The test data are identified by the “verification” and the associated “test and verification” (T&V) software description. The coverage analysis is identified by the “conclusion” associated with each test and source of the “notes” of the T&V software description.

The complete understanding of the test plans presented below requires some knowledge of ring mechanisms. A good description of the SCOMP ring mechanism can be found in [24]. In the example presented below, the following abbreviations have been used:

- Ring numbers are represented by the contents of registers R_0 – R_3 such that $R_0 \leq R_1 \leq R_2 \leq R_3$, and $0 < R_i \leq 3$.
- $R_{eff} = \max(R_{curr}, R_{caller})$ is the effective ring of the accessor, where R_{curr} is the current ring of execution and R_{caller} is the ring of the caller program.

- The offset is the entry point into the segment of the called program. (offset = 0)
- $R_{from}, (R_{to})$ is the ring from (t_o) which control is transferred with $R_{to} = < R_{from}$ calls and $R_{to} \geq R_{from}$ for returns.
- The T register contains the segment number of the stack segment associated with the current ring.

3.7.6.1 Test Conditions for the Ring Crossing Mechanism

- (1) Test that the ring-call mechanism changes the ring numbers such that $R_{to} \leq R_{from}$ transfers to entry point zero (offset = 0) of the called-program segment and requires that the execute bit is turned on in the descriptor for the called-program segment.
- (2) Test that the ring-return mechanism changes the ring numbers such that $R_{to} \geq R_{from}$.
- (3) Test that each ring is associated with a different per-ring stack segment after the ring call/return is made.

3.7.6.2 Test Data

(1) Environment Initialization

The following sections of the T&V software descriptions T200 and T1100 contain the environment initialization used by the test programs which invoke the SCOMP call (LNJR) and return (RETN) instructions.

Test and Verification Software Description

T200 TCALL (Test Call and Return Instructions):

- A. Execute return and call instructions between two rings to test a single ring change.
- B. Test multilevel ring change. Change rings from ring 0 to ring 3 (one ring at a time), return to ring 0 in reverse order.

C. Test transfer of *T* register data on ring changes.

ALGORITHM:	TEST NUMBER:
Put known values in <i>T</i> registers	
Return to ring 3 (TCALL3)	200
Call to ring 0 (TCALL)	201
Return to ring 1 (TCALL1)	202
Return to ring 2 (TCALL2)	203
Return to ring 3 (TCALL3A)	204
Check <i>T</i> register	205
Call to ring 2 (TCALL2)	206
Check <i>T</i> register	207
Call to ring 1 (TCALL1)	208
Check <i>T</i> register	209
Call to ring 1 (TCALL)	20A
Check <i>T</i> register	20B

Notes:

1. Halt on failures, identify the test failed.
2. For call, set $R \leq R_{eff} \leq R_3$, offset = 0, and execute permission is "on."
3. Negative tests are not required, these are tested under Trap Handling (namely, T1100 below).
4. Identify controlling descriptors for each test.
5. Inputs identify supporting code and data locations and controlling descriptors.
6. Separate blocks are shown in the structure chart since the process is spread across three rings.

Outside Services Required: None.

Test and Verification Software Description

T1100 TRING (Test Ring Traps)

Execute call using descriptors with the following trap conditions:

- A. $R_{eff} > R_3$, segment offset $\neq 0$ (use page offset 0), execute permission off.
- B. Execute return with $R_{t0} < R_{eff}$.

ALGORITHM:	TEST NUMBER:
Change to ring 1	
Call - E_{off}	1100
Call - Offset NEO	1101

TEST PLAN EXAMPLES

Call - R_{eff} GT $R_3(0)$	1102
Return - R_{eff} GT R_{t_0}	1103

ALGORITHM:

TEST NUMBER:

Change to ring 2	
Call - R_{eff} GT $R_3(0)$	1104
Call - R_{eff} GT $R_3(1)$	1105
Return - R_{eff} GT $R_{t_0}(0)$	1106
Return - R_{eff} GT $R_{t_0}, (1)$	1107
Change to ring 3	
Call - R_{eff} GT $R_3(0)$	1108
Call - R_{eff} GT $R_3(1)$	1109
Call - R_{eff} GT $R_3(2)$	110A
Return - R_{eff} GT $T_{t_0} (0)$	110B
Return - R_{eff} GT $T_{t_0}, (1)$	110C
Return - R_{eff} GT $T_{t_0} (2)$	110D

Change to ring 0 via seg 4

Notes:

1. Halt on failure; identify test failed.
2. Identify controlling descriptors for each test.
3. Provide trap handler that verifies SPM hardware trap functions and recovers from the trap. Correct functioning should render the expected trap transparent.
4. All of the tests are executed in ring 3.

Outside Services Required: SI - Trap Handler (TH14)

(2) Test Parameters

The test programs require no input for these test conditions. The outputs of the test program represent the test outcomes defined below.

(3) Test Outcomes

Outcomes for Test Condition (1)

- Success: $R_1 \leq R_{eff} \leq R_3$ and offset = 0 and E privilege = OFF; (namely, test numbers 201, 206, 208, and 20A).
- Failure: $R_3 > R_{eff}$ or offset $\neq 0$, or E privilege = OFF; (namely, test numbers 1102, 1104–1105, 1108–1110A, or 1101, or 1100).

Outcomes for Test Condition (2)

- Success: $R_1 \geq R_{eff}$ (namely, test numbers 200, 202, 203, and 204).
- Failure: $R_{t_0} < R_{eff}$ (namely, test numbers 1103, 1106, 1107, 110B–110D).

Outcomes for Test Condition (3)

- Success: T registers contain the stack segment number placed in there in T200. This outcome is obtained for test numbers 205, 207, 209, and 20B.
- Failure: This outcome is not expected.

3.7.6.3 Coverage Analysis

The test conditions (1)–(3) above have been derived from descriptions of the SCOMP processor and of the Security Protection Module (SPM). The SCOMP FTLS of the user-visible hardware functions were either incomplete or unavailable at the time of processor security testing and, therefore, could not be fully used for the generation of test conditions [3]. Since a formal model of the protection mechanisms of the SCOMP processor was unavailable, the documentation of the SCOMP processor and SPM were the only available sources of test conditions.

The test coverage analysis for the conditions (1)–(3) above is based on boundary value coverage. Note that test condition (1) includes three related subconditions, namely (offset = 0) and (E privilege = ON). Furthermore, subcondition ($R_0 < = R_{eff} <$) requires at least three calls (i.e., R_3 to R_0 , R_3 to R_1 , R_3 to R_2) be made and that each be combined with subconditions (offset = 0) and E privilege = ON). Though subcondition $R_3 > R_{eff}$ requires that six calls be made (i.e., for $R_{eff} > R_3 = 0$, $R_{eff} > R_3 = 1$, $R_{eff} > R_3 = 2$), these subconditions cannot be combined with subconditions (offset \neq 0) and E privilege = OFF) because all these related subconditions return failure. Test condition (2) similarly requires that multiple calls be made. It should be noted that for test condition (3) the boundary-value coverage can only cover the success subcondition in normal mode. The lack of a current stack segment number in the T register after a call or a return could only happen due to processor or SPM failures.

Several test conditions may be desired for processor security testing for which test programs cannot be built in the normal mode of operation. The example of this is provided by the invalidation of current process descriptions in the processor cache before dispatching the next process. (A complete test of the invalidation

function for descriptions in the cache can be performed in privileged mode or in ring 0 as outlined in the conclusions below.)

Test Conditions for Descriptor Invalidation

Test that the descriptors contained in the cache are invalidated prior to the dispatch function.

Test Data

A test to insure invalidation of SPM descriptors after dispatch is not in the test software.

Verification by Analysis

The invalidation function of dispatch involves resetting of all SPM cache validity bits for direct memory descriptors used by the CPU. This requires invalidation of 256 and 64 cache locations in the Virtual Memory Interface Unit (VMIU) and Descriptor Store boards, respectively. Analysis has confirmed the proper implementation of this function.

Conclusions

A test to verify the dispatch invalidation function could be implemented by using two descriptor structures, each mapping CPU memory references to different areas of memory. By checking usage (U and M bits) of each direct memory descriptor and the actual access to different locations in memory, the invalidation of previously stored descriptors in the SPM cache could be determined. The VMIU portion of the test would use a page descriptor structure (256 page located within 16 contiguous segments) with checks provided for each page of memory. The descriptor store board portion of the test would be constructed in a similar manner except 64 direct segment descriptors would be used.

3.7.7 Relationship with the TCSEC Requirements

In this section we present the documentation requirements for test plan and procedures stated by the TCSEC and additional recommendations derived from those requirements. Responsibility for documenting test plans and procedures belongs both to evaluators and to vendors because security testing evidence is

produced by both for different purposes. It should be noted that the evaluator's test documentation and programs will not fulfill the vendor responsibility for providing test documentation and test programs. Wherever appropriate, this section differentiates exclusive evaluator responsibility from that of the vendors. Citations of specific evaluator responsibility provided by the TCSEC are omitted here because they are explained in detail in Section 10 of reference [13].

The introductory section of the test documentation should generally identify the product to be evaluated and give a high-level description of the system being tested, the types of applications for which it is designed, and the evaluated product class for which it is being tested.

PURPOSE AND SCOPE OF SECURITY TESTING BY EVALUATORS

A section should state the objectives of security testing conducted by the vendor and describe the role that this testing plays in supporting the Technical Evaluation Phase by the NCSC. It should state the purpose of the test plan and how it will be used. It should also define the intended scope of the effort in terms of both hours of effort and elapsed time, as well as the technical scope of the effort.

ROLES AND RESPONSIBILITIES OF SYSTEM EVALUATORS

A section should describe how the test team is organized and identify the team leader and all members by name and organization, qualifications, and experience. Its purpose is two fold. First, it should clearly delineate team members' responsibilities and relationships. Second, it should provide sufficient background information on the team's prior functional testing experience to substantiate that the team is qualified to conduct the tests. It should describe the level of previous experience that each team member has with the system being evaluated, whether all team members have completed an internals course for the system, how well the team understands the flaw hypothesis penetration testing methodology and vulnerability reporting process, and other relevant qualifications. This section should specifically address test team education, skill, and experience.

A section should also identify any responsibilities for coordination, review and approval of the test plan, and procedures and reports that lie with personnel outside the test team.

SYSTEM CONFIGURATION

A section should specify the hardware and software configuration used for testing to include the location of the test site. This configuration should be within the configuration range recommended by the vendor for approval by the NCSC during the Vendor Assistance Phase. The vendor will be required to identify and recommend a test configuration to the NCSC early in the evaluation process. The vendor's recommendation will be considered by the NCSC test team in selecting the "official" test configuration.

Hardware Configuration

A subsection should identify the CPU, amount of random access memory (RAM), I/O controllers, disk drives, communications processors, terminals, printers, and any other hardware devices included in the test configuration by specifying the vendor's model number and quantity of each configuration item. Each peripheral should be given a unique identifier that associates it with a specific controller port. Communications parameter settings should be specified where appropriate. It should be possible to duplicate the test configuration exactly from the information provided.

Software Configuration

A subsection should identify the version of the vendor's operating system included in the test configuration, as well as each specific TCB software component that is not part of the operating system. It should include sufficient information to generate the system from the TCB test software library along with the vendor's distribution tapes. It is very useful to include a summary of device driver file names and the file system directory structures along with a description of their general contents.

SECURITY TEST PROCEDURES (TO BE FOLLOWED BY BOTH VENDORS AND EVALUATORS)

The TCSEC states the following test documentation requirement:

Class C1 to A1. "The system developers shall provide to the evaluators a document that describes the test plan, test procedures

that show how the security mechanisms were tested and the results of the security mechanisms' functional testing."

A section should provide both an overview of the security testing effort and detailed procedures for each of the security test plans. Security testing will include detailed procedures for executing any test plan that is needed to provide significant coverage of all protection mechanisms. This portion of the test plan must be detailed; it will require the test team to generate the test plans for each TCB primitive.

Review and Evaluation of Test Documentation for Each TCB Primitive

A subsection will present an evaluation of the method of TCB primitive testing used by the vendor's development team, the completeness of the coverage provided by the vendor's tests for the TCB primitive, and any shortfalls that will need to be covered by the security testing team. This evaluation should include a discussion of the extent to which the vendor's tests used black-box (which does not necessarily assume any knowledge of system code or other internals) or gray-box coverage (which assume knowledge of system code and other internals). Black-box test coverage is best suited for C1 to B1 class systems. Gray-box coverage of a system's security protection with respect to the requirements in the *TCSEC* is best suited for B2 to A1 class systems. In terms of TCB-primitive coverage, this subsection should identify any relevant interfaces or mechanisms that the vendor has previously failed to test, as well as how thoroughly the vendor has previously tested each interface or mechanism with respect to each *TCSEC* requirement.

Test Plans

Test Conditions

This section identifies the test conditions to be covered. It should include explanation of the rationale behind the selection and the order in which the tests are executed. It is recommended that the detailed test procedures for each test condition be compiled in annexes in a format that enables the test personnel to mark steps completed to ensure that procedures are performed correctly.

These test conditions should be derived from interpretations of the following:

- Protection philosophy and resource isolation constraints (for classes C1 and C2).
- Informal security models (class B1).
- DTLS and formal security models (classes B2 and B3), FTLS (class A1).
- Accountability requirements (all classes).

Test Data

The test data should include the definition of the following:

- Environment initialization.
- Test parameters.
- Test outcomes.

Coverage Analysis

The coverage analysis section of a test plan should *justify the developer's choice* of test conditions and data, and should delimit the usefulness of the test with respect to security of the system.

Test Procedure Format

Whenever security testing is not automated extensively, the developer's test documentation should include test scripts. These should contain:

- A description of the environment initialization procedure.
- A description of the execution test procedure.
- A description of the result identification procedure.

Procedure for Correcting Flaws Uncovered During Testing

A subsection should describe the procedure for handling the identification and correction of flaws uncovered during the course of functional testing. It should specify how this information was provided to the vendor's test team, how much time

was allocated to correct the flaw, and how testing again was conducted to verify that flaws have been corrected.

An Example Test Report Format

The *TCSEC* includes the following requirements for reporting the test results:

Classes C1 to A1. "The system developer shall provide to the evaluators a document that describes [the] results of the security mechanisms' functional testing."

A section should identify the vendor of the evaluated product and give a high-level description of the system that was tested, the types of applications for which it is designed, and the class for which it is being evaluated.

Test System Configuration

A section should provide a general description of the test system configuration. It need not be as detailed as the test plan, but should give enough detail to identify the hardware and software context for the test results.

Test Chronology

A section should provide a brief chronology of the security testing effort. It should indicate when testing began, where it was conducted, when each milestone was completed, and when testing was completed.

Results of Security Testing

A section should discuss each flaw uncovered in the system during security testing. It should describe any action taken to correct the flaw as well as the results of retesting. It may be useful to define a "level of criticality" for classifying the flaws in order to distinguish major problems that might impact the final rating from minor discrepancies or those for which a work-around was found.

List of Uncorrected Flaws

A section should identify any problems that were uncovered during testing that were not corrected to the test team's satisfaction.

4. COVERT CHANNEL TESTING

Covert channel testing is required in order to demonstrate that the covert channel handling method chosen by system designers is implemented as intended. These methods include prevention and bandwidth limitation. Testing is also useful to confirm that the potential covert channels discovered in the system are in fact real channels. Testing is also useful when the handling method for covert channels uses variable bandwidth-reduction parameters (e.g., delays) that can be set by system administrators (e.g., by auditors). Testing can ensure that these mechanisms reduce the channel bandwidths to the correct limits intended by system administrators.

Bandwidth estimation methods that are necessary for the handling of covert channels may be based on engineering estimation rather than on actual measurements. Bandwidth estimations provide upper bounds for covert channels before any handling methods are employed. In contrast, covert channel testing always requires that actual measurements be performed to determine the covert channels' bandwidths after the chosen handling method. Similarly, whenever covert channels are prevented (i.e., eliminated), testing of actual code of the implemented system is required.

4.1 COVERT CHANNEL TEST PLANS

The test plans used for covert channel testing have the same structure as those used for security functional testing. That is, for testing each covert channel a test condition and the test data should be written, and coverage analysis should be performed for that channel.

The test conditions for channels differ depending on the choice of the covert channel handling method. Test conditions would state that no information leakage is possible through the previously extant channel for covert channels that are eliminated. For covert channels handled by bandwidth limitation, the condition would state that the measured bandwidth of the respective channel is below the predicted limit chosen for the channel. If the test is used to measure the bandwidth after nonzero delay values are used, the predicted bandwidth limit is the target bandwidth chosen or provided by the default values of the added delays. If the test is used to

measure the bandwidth before nonzero delays are used, the predicted bandwidth is the estimated maximum bandwidth of each channel.

The test data for each channel consists of the test environment definition, the test parameters used, and the outcomes of the test. The test environment definition consists of a description of the actual covert channel scenario defining how the sender and recipient processes leak information. This definition may include a description of the synchronization methods used by the sender and receiver, the creation and the initialization of the objects (if any) used by the sender/receiver to leak information, the initialization and resetting of the covert channel variable, etc. If channels are aggregated serially or in parallel, and if specific encodings are used, the aggregation and encoding methods should be defined. (Note that neither channel aggregation nor special bit encodings need to be used in testing as these are neither required nor recommended by either [13] or its covert channel guidelines.)

It should be noted that in many cases of resource exhaustion channels, the test program need not actually leak any string of bits. This is acceptable only in cases when the exhaustion of one of these resources deteriorates system performance to such an extent that no information could possibly be transmitted within 1 second. In such cases, it is sufficient to measure the elapsed time from the beginning of the covert channel primitive invocation until the resource exhaustion error is returned to test the upper bound of the achievable bandwidth.

The test parameters consist of the values set to run the measurements and include the number of bits leaked for each channel, the distribution of 0's and 1's chosen for the test, the representation of 0's and 1's (e.g., using the states of the covert channel variable or system objects), the delay values chosen for testing, the number of objects used and their types and privileges, etc.

The test outcomes specify the expected results of the test. As with test conditions, the test outcomes are similar for all channels. For each channel, they define the target limit of the actual, measured channel bandwidth.

Coverage analysis for covert channel testing requires the demonstration that the placement of delays and randomization points in code covers all information flow

paths. Credible assurance of correct handling of covert channels cannot be provided without such analysis.

To understand the complexity of covert channel testing and the need for covering all information flow paths, consider a generic example of covert channels provided by a single variable. Assume that the variable can be viewed (altered) by V (A) primitive calls of the TCB. These primitives can create up to $V \times A$ covert channels. Testing would have to ensure that, if the covert channel handling method is based on placement of bandwidth reduction delays, the placement of those delays limits bandwidth of all these covert channels to a specified value. In a system that has N variables (or attributes) that create covert storage channels, $\sum_{i=1}^N (V_i \times A_i)$ test programs would have to be generated to assure correct placement of delays. For a UnixTM-like system, this would require approximately 3,000 test plans and programs for covert channel testing, many of which would be redundant. This would clearly be impractical.

The assurance that covert channel tests cover all possible flows in the TCB can be provided by (1) a test of a single instance of covert channel usage, which would test that the channel is eliminated or delayed, and by (2) an analysis that shows that all other instances of the same channel are either eliminated or have their bandwidth reduced by correct placement of delays. This assurance allows the elimination of all redundant covert channel tests without loss of coverage.

4.2 AN EXAMPLE OF A COVERT CHANNEL TEST PLAN

In this section we present an example of a test plan for a covert storage channel. This channel, called the Upgraded Directory Channel, has been described in references [15] and [16]; therefore, it will not be described here in detail. Measurements and engineering estimations, which predict the bandwidth of this channel in Secure XenixTM running on a 6 megahertz personal computer AT, have been reported in reference [16]. Other types of engineering estimations which can determine the maximum bandwidths of noiseless covert channels have been presented in reference [21].

See Section 4.3, "Relationship with the TCSEC Requirements," which contains an example of a covert channel test plan.

4.2.1 Test Plan for the Upgraded Directory Channel

System Version: PS/2 Model 80.

Covert Channel Type: MAC conflict channel [19] .

Variable Name: direct -> d_ino.

4.2.1.1 Test Condition

The test condition for the "upgraded directory channel" is:

The measured bandwidth is lower than the predicted bandwidth after delay is added.

4.2.1.2 Test Data

Environment Initialization

The test operator logs in at a security level called "Low" and initializes a receiver process. Then the operator logs in at a level called "High" and initializes a sender process. Level High must dominate level Low. The receiver process creates an upgraded directory at level High and the sender process, which is at the same level as that of that directory, signals to the receiver process a 1 or a 0 by either creating or not creating an object in that directory. The receiver process may detect 0s and 1s by trying to remove the upgraded directory. Success or failure of the removal operation signals 0s or 1s because a directory can only be removed when there is no object in that directory [15].

Note: Both the sender and the receiver use four directories to amortize the synchronization and environment set up delay for every bit over four bits (i.e., four-bit serial aggregation). This covert channel scenario is shown in Figure 7.

Scenario of Use for the Upgraded Directory Channel

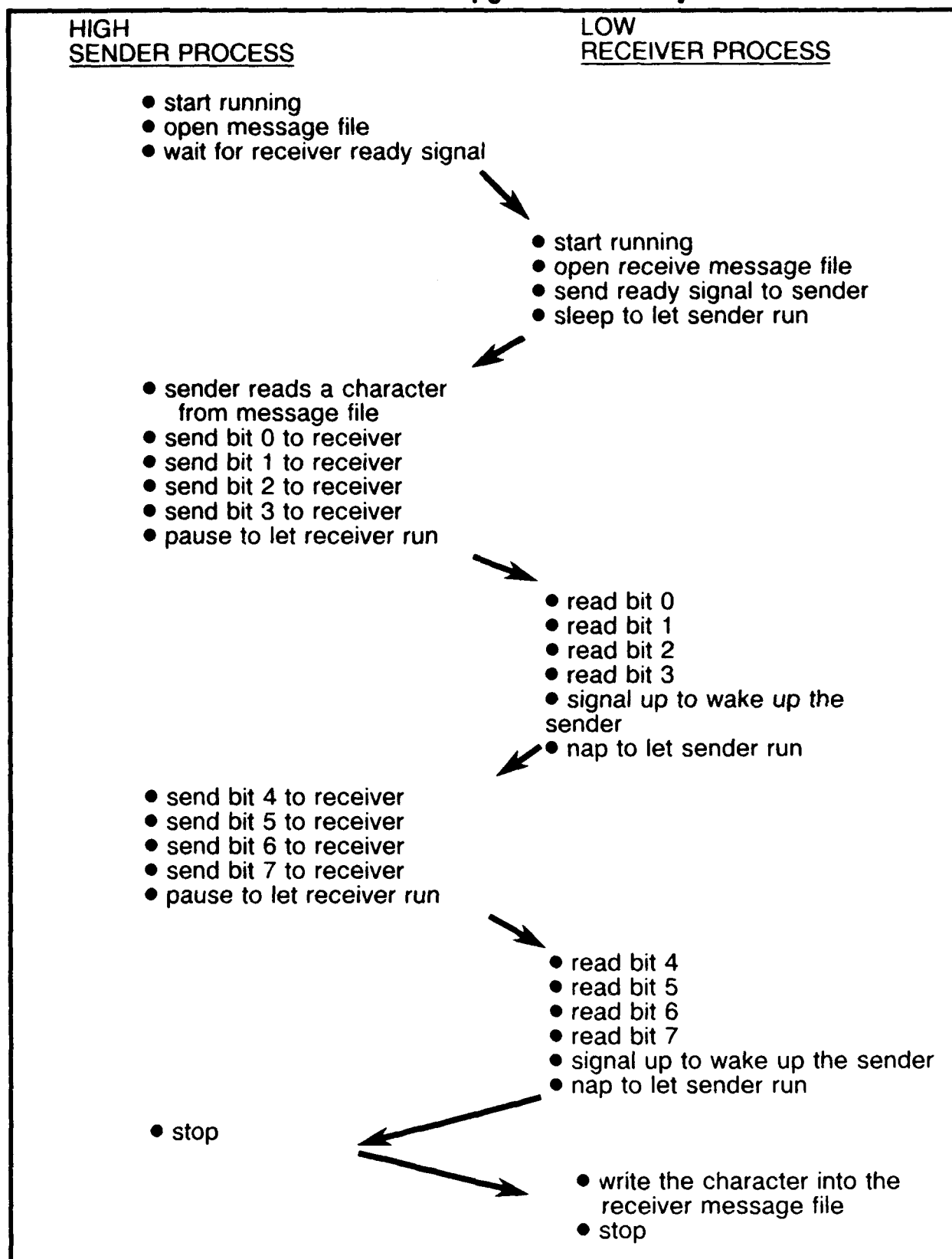


Figure 7

Parameters

Number of Bits leaked: 8.

Distribution of information used by the test program: 01100011, which represents character "c."

Object Type: directory.

Number of objects used: 4 directories (serial four-bit aggregation).

Measurements: The `rmdir` (nonempty directory) elapsed time is 3020 ms. (delayed). The `rmdir` (empty directory) elapsed time is 180 ms. The `rmdir` (average) elapsed time is 1600 ms.

Outcome: The measured bandwidth is less than the predicted bandwidth of 0.566 bit/sec (with delay). If delay is removed, the predicted bandwidth is 2.8 bits/sec.

4.2.1.3 Coverage Analysis

The trusted process `rmdir` is the only primitive that reads variables in this covert channel.

4.2.2 Test Programs

The test programs are included in files `up/dirs.c` and `dirr.c` (not shown here).

4.2.3 Test Results

The measured bandwidth is 0.5 bit/sec. The reason the test results for the "no delay" case are not included here is that this delay is built into the system configuration. The auditor cannot turn off or set the delay.

4.3 RELATIONSHIP WITH THE TCSEC REQUIREMENTS

The TCSEC states the following requirement for the documentation of covertchannel testing:

Classes B2 to A1. "The system developers shall provide to the evaluators a document that ... shall include the results of testing the effectiveness of the methods used to reduce covert channel bandwidths."

To satisfy this requirement the testing of the covert channel bandwidth must be performed. The following format is recommended for the documentation of covert channel test plans.

(1) Test Conditions

These conditions should be derived from covert channel handling methods and should include:

- Elimination of covert channel conditions (whenever appropriate).
- Bandwidth limitation conditions based on measurements or engineering estimations.
- Covert channel audit conditions (where appropriate).

(2) Test Data

Test data should include:

- Environment initialization data and/or a brief scenario of covert channel use;
- Test parameter definition.
- Test outcome (a blocked channel, an eliminated channel, or measured bandwidth below the predicted value).

(3) Coverage Analysis

This analysis should contain an explanation of why the test covers all modes of covert information leakage through an individual channel, through a channel variable, or through a class of channels.

5. DOCUMENTATION OF SPECIFICATION-TO-CODE CORRESPONDENCE

The correspondence of the formal specification and source code is also a test documentation requirement of the *TCSEC*. The test documentation requirements of the *TCSEC* state:

Class A1. "The results of the mapping between the formal top-level specification and the TCB source code shall be given."

This A1-exclusive requirement is only peripherally related to security testing. We have only included it as an appendix for the interested reader. The detailed set of FTLS-to-code correspondence requirements is provided by *A Guideline to Formal Verification Systems* (NCSC-TG-014).

APPENDIX

Specification-to-Code Correspondence

1. Overview

The requirements of the FTLS-to-code correspondence stated in Section 5 define the scope of the mapping process. The mapping may be informal but it must:

- Show the consistency of the TCB implementation code and FTLS.
- Include all elements of the FTLS in the correspondence.
- Describe code not specified in the FTLS (if any) excluded from the mapping.

Although the mapping may be informal, it is instructive to review its theoretical underpinnings. These underpinnings are summarized in Sections 1–4 of reference [25] and in Sections III and IV of reference [26].

Consider two specifications of a finite state machine M denoted by M_f and M_c . The specification M_f is the FTLS of M , and M_c is the implementation specification (i.e., code) of M . A common thread of all formal verification methods is the requirement to demonstrate that any state of the machine specification M_c represents a state of another, more abstract, machine specification M_f . Alternate methods exist that attempt to formally establish this representation.

The first method is based on defining a function ϕ with an application to a state S_c of M_c that yields the “corresponding” state of M_f . The function F defines the mapping between the two machine specifications. This mapping expresses properties of the correspondence between M_c and M_f . For example, if the property of M_c is to mimic M_f step by step, the mapping function F should be defined in such a way that the i -th state of M_f corresponds to the i -th step of M_c . If the notions of a secure state and state transition are defined in M_f , and if all state transitions of M_f leave it in a secure state, the property of the function F is defined in such a way that all mapped states of M_c are secure and all state transitions of M_c leave it in a

secure state. In general, the mapping function F should capture the specific property, or properties, desired for the mapping.

The first mapping method, called "refinement mapping" in reference [25], is applicable to large classes of problems of establishing code-to-specification correspondence, including correspondence of concurrent program code. In many cases, however, the refinement mapping cannot be found. Reference [25] shows that in a very large class of mapping cases it is possible to augment the implementation specification of M_C (i.e., the code) with extra state components (called the "history" or "prophecy" variables) in a way that makes it possible to produce refinement mapping.

The second method is based on defining a function G whose application to an assertion A_f defined on a state of M_f yields an assertion A_c about the state of M_C . This alternate mapping should also capture similar properties of M_f and M_C as those defined above. These two notions of mapping defined by F and G are inverses of each other, as argued in reference [16], in the sense that:

For any assertion A_f about the states of M_f and any state S_C of M_C , A_f is true of state $F(S_C)$ of M_f if and only if assertion $G(A_f)$ is true of state S_C .

Examples of how the two mapping definitions are applied to system specifications and design are provided in [26]. A further example, which uses similar methods for the generation of correct implementation code from abstract specifications, is given in [27]. In both references, the mappings are defined on types, variables, constants, operators (e.g., logic operators), and state transformations. The common characteristics of all formal mappings are (1) the mapping definition, (2) the identification and justification of unmapped specifications (if any), (3) the specification of the properties that should be preserved by the mappings, and (4) the proofs that these properties are preserved by the defined mappings.

2. Informal Methods for Specification-to-Code Correspondence

Informal methods for FTLS-to-code correspondence attempt, to a significant degree, to follow the steps prescribed by formal methods. Two informal exercises of FTLS-to-code correspondence are presented briefly in references [28 and 29], one based on FTLS written in SPECIAL and the other in Ina Jo. Analysis of both

exercises, one of which was carried out on the SCOMP kernel [28], reveals the following common characteristics.

2.1 Mapping Definition

The mapping units of both FTLS and code are identified and labeled explicitly. For example, each "processing module" is identified both in FTLS and code. This identification is aided by:

- Intermediate English language specification or program design language specifications, and/or
- Naming conventions common to FTLS and code (if common conventions are used). Processing modules are represented by the "transform" sections of Ina Jo and by the module V, O, and OV functions of SPECIAL.

Alternatively, the mapping units may consist of individual statements of FTLS and implementation code.

Correspondences are established between similarly labeled mapping units. This is particularly important for the units that have user visible interfaces. Correspondences include:

- Data structures used by processing modules (namely variables, constants, and types of the FTLS) are mapped in their correspondent structures of code.
- Effects of processing modules and operators (e.g., logic operators) that are mapped to the corresponding code functions, procedures, statements and operators.

In addition, whenever the effects sections of a processing module identify exceptions separately, the correspondence of FTLS exceptions and code exceptions should also be included explicitly.

2.2 Unmapped Specifications

The process of establishing the FTLS-to-code correspondence may reveal that the FTLS has no corresponding code or has incomplete code specifications. This situation is explicitly excluded by the TCSEC requirements, because all elements of

the FTLS must have corresponding code. More often, significant portions of the implementation specifications (i.e., code) remain unmapped. Mismatches between FTLS and implementation specification may occur for many different reasons, which include:

- FTLS and code are written in languages with very different semantics. This is the case whenever FTLS are written in nonprocedural languages and code is written in a procedural language. In this case, the correspondence between the assertions of the nonprocedural language and the functions, procedures, and control statements of the procedural language are difficult to establish. Some unmapped implementation code may represent implementation language detail which is not mapped explicitly to FTLS and which does not affect adversely the properties preserved by the mapping (discussed below).
- The domain or range of an FTLS function may be incorrectly identified during code development. In this case the mapping process should be able to identify the cause of the FTLS and implementation code mismatch.
- A significant part of the TCB code is not visible at the user interface and thus, has no correspondent FTLS. This code, which includes internal daemons, such as daemons for page/segment replacement, daemons that take system snapshots, and so on, is nevertheless important from a security point of view because it may introduce information flows between TCB primitives in unexpected ways. The effect of such code on the mapping, or lack thereof, should be carefully analyzed and documented.
- Unmapped TCB implementation code includes code which ensures the noncircumventability and isolation of the reference monitor and has no specific relevance to the security (i.e., secrecy) policy supported. For example, TCB implementation code which validates the parameters passed by reference to the TCB is policy independent and may cause no covert channel flows (because all relevant flows caused by these checks are internal to the process invoking the TCB).
- Unmapped TCB implementation code may include accountability relevant code (e.g., audit code), debugging aids, and performance monitoring code, as well as other code which is irrelevant to the security supported. The

presence of such code within the TCB may introduce information flows within the TCB primitives and may introduce additional covert channels. The effect of such unmapped code on the mapping should be analyzed and documented.

- The TCB may contain implementation code that is relevant to the security policy supported by the system but irrelevant to the properties that could be verified using the FTLS. For example, the correctness of some of the discretionary access control policies may not be easily verified with the currently available tools. Therefore, the complete mapping of code implementing such policies to the corresponding FTLS may have limited value. However, the information flows generated by such code should be analyzed and documented.

2.3 Properties Preserved by the Mapping

A key characteristic of any FTLS-to-code mapping is the specification of the security property of the FTLS that should be included in implementation code. Such security properties include specifications of MAC and DAC policy components, object reuse components, and accountability components, all of which are user visible at the TCB interface. These properties should also include specifications of equivalence between information flows created by FTLS and those created by implementation functions, procedures, variables and mapped code. Other safety properties and liveness properties may also be included. For each mapped module, the properties preserved by that module should be documented.

It must be noted that current emphasis of practical work on FTLS-to-code mapping is exclusively focused on the maintenance of (1) mandatory access control properties of FTLS in implementation code, and (2) the equivalence between covert channel flows of the FTLS and those of the implementation code.

2.4 Correlation Arguments

The documentation of each correspondence between mapping units should include a convincing argument that the desired properties are implemented by code despite unmapped specifications or code (if any). Lack of such documentation would cast doubts on the validity of the mapping and on the usefulness of demonstrating formally such properties of FTLS. For example, little use is made of

the soundness of information flows of FTLS whenever flow equivalence between FTLS primitives and variables and those of implementation code is not established.

3. An Example of Specification-to-Code Correspondence

The module whose FTLS mapping to implementation code is illustrated in this section is "get_segment_access" system call of the Honeywell's Secure Communication Processor (SCOMP). The FTLS is written in SPECIAL, the language supported by the Hierarchical Development Methodology (HDM) developed at SRI International, and the implementation code is written in UCLA Pascal. The system call "get_segment_access" returns the access privileges of the invoking process (e.g., user process) for a uniquely identified segment. The effect of this call is similar to that of the **access** system call of UnixTM when applied to files, namely Tables 1 and 2, page 58. Figure 8 below shows the FTLS of "get_segment_access" and Figures 9a; 9b, parts 1 and 2; and 9c show its implementation code. Note that Figure 9a identifies the "def.h" file, namely the file of included header definitions of the module, Figure 9b, parts 1 and 2, contain the actual code of the module (i.e., in the ".p" file), and Figure 9c contains the code of implementation function "get_segment_info," which is invoked by the code of "get_seg_access."

3.1 Mapping Definition

Mapping Units

The mapping units for both the FTLS and the implementation code of SCOMP are the individual language statements. To establish the mapping each statement of the implementation code is labeled unambiguously (i.e., using the code or data file name and the statement line number). Statement level labeling of data definitions (i.e., "def.h" files) and code (i.e., ".p" files) is shown in Figures 9a; 9b, parts 1 and 2; and 9c.

Correspondence of Labeled Units

The statement level mapping of FTLS to code is established in SCOMP by adding to each SPECIAL statement of the "get_segment_access" module the corresponding individual (or group of) UCLA Pascal statement(s). Figure 8 shows this.

FTLS Written in SPECIAL for "get__segment__access"

TLS to Code Mapping	Kernel Software
[def.h 370, 372]	<pre> VFUN get__segment__access (unique__id seg__name) {unique__id proc; access__level pl} -> access__info seg__access; \$(returns access attributes of segment) </pre>
[segment.p 907-908]	EXCEPTIONS
[segment.p 909-923, 931-939]	<pre> invalid__segment__name : invalid__partition(seg__name) OR object__type(seg__name) != seguid; segment__does not exist : ~(sys__obj__info(seg__name, level__of(seg__name)) = exists AND filesystem(segment__filesystem(seg__name, level__of(seg__name)), level__of(seg__name)), mounted AND valid__flow(level__of(seg__name), pl)); \$(above valid flow statement used instead of following for proof - non__discretionary__allowed(proc, seg__name, {read}, level__of(seg__name))) </pre>
[segment.p 914-917, 945-978]	DERIVATION
	<pre> get__object__access(seg__name, level__of(seg__name)); </pre>

Figure 8

LEGEND

- ^ = pointer to
- != = not equal
- ~ = negation

Implementation Code Written in UCLA Pascal for "get_seg_access"

```

370      function get_seg_access
371      (var      seg_access_p      :access;
372      var      seguid_p          :seguid;
373      ring_p      :word)      :error_code      forward;      {segment}
374
375      function get_segment_info
376      (var      table_type_p      :seg_table_type;
377      var      branch_p          :branch_table;
378      mnt_inx_p      :mount_index;
379      var      seguid_p          :seguid)      :~branch_table      forward;      {segment}
380

```

Fragment for the "def.h" File for Inclusion in "get_seg_access"

Figure 9a

User Visible Effects, Exceptions, and Data Structures

The only user visible effect of this VFUN (i.e., state returning function) is mapped to the language statements "segment.p 931-939" as part of the nondiscretionary access check performed to determine whether the calling process has MAC access to the segment passed as a parameter. Whenever this check is passed (in "segment.p 931"), the accesses of the caller process to the segment are returned through "seg_access_p" parameter. Note that the UCLA Pascal function "non_discretionary_access_allowed(...)" is mapped to the SPECIAL function "valid_flow(...)." The former calls the UCLA Pascal version of the latter (neither shown here). In addition, the function "non_discretionary_access_allowed(...)" also performs checks to determine whether the invoking process has special system privileges that would allow it to bypass the MAC checks of "valid_flow(...)." Since the properties of interest to the FTLS verification do not include the effects of the system privileges, only the SPECIAL function corresponding to "valid_flow(...)" is used in the VFUN "get_segment_access" (namely, comment in the FTLS). Note that the derived SPECIAL function "get_object_access" corresponds to the UCLA Pascal function "get_segment_info," defined in "segment.p 945-978" and invoked in "segment.p 914-917," and that both are invisible at the TCB interface when used in the corresponding modules "get_segment_access" and "get_seg_access."

Implementation Code Written In UCLA Pascal for "get_seg_access"

```

887
888  Function  get_seg_access(      var      seg_access_p  :access;
889                                var      seguid_p      :seguid;
890                                ring_p      :word}      :error_code;
891
892  var
893      temp_seguid_l      :uid;
894      register          mnt_inx_l      :integer;
895                        table_type_l   :seg_table_type;
896                        branch_ptr_l   :branch_table;
897                        aste_ptr_l     :aste;
898                        mode_l         :access_mode;
899
900  begin
901      #ifdef  TFLAG
902      if ((bit_test(tracing, TKERNEL)) | (bit_test(tracing, TFUNCALL)))
903          then trace("qtsqccss");
904      #endif
905      if !fublock(seguid_p, temp_seguid_l, ring_p, size(temp_seguid_l))
906          then return(invalid_argument_pointer);
907      if !in_range(temp_seguid_l.uid_id[0], min_seg_partition, max_seg_partition)
908          then return (invalid_segment_name);
909      mnt_inx_l := find_mte(temp_seguid_l.uid_filesys);
910      if (mnt_inx_l = BAD_MTE)
911          then return(segment_does_not_exist);
912      inc(mount_vector[mnt_inx_l].mte_seg_count);
913      lock_read(segment, temp_seguid_l);
914      branch_ptr_l := get_segment_info(table_type_l,
915                                     temp_branch_l,
916                                     mnt_inx_l,
917                                     temp_seguid_l);
918      if branch_ptr_l = NULL
919          then begin
920              unlock_read(segment, temp_seguid_l);
921              dec(mount_vector[mnt_inx_l].mte_seg_count);
922              return(segment_does_not_exist);
923          end;
924      if (table_type_l = aste_type)
925          then begin
926              aste_ptr_l := branch_ptr_l;
927              temp_branch_l.bt_sbte_access := aste_ptr_l .aste_sbte.sbte_access
928          end;
929      unlock_read(segment, temp_seguid_l);
930      dec(mount_vector[mnt_inx_l].mte_seg_count);
931      if !non_discretionary_access_allowed(temp_branch_l.bt_sbte.sbte_access, READ_ACCESS)
932          then begin
933              mode_l := READ_ACCESS;
934              audit_event(segment_access_violation,
935                          addr(temp_seguid_l),
936                          addr(temp_branch_l.bt_sbte.sbte_access),
937                          addr(mode_l));
938              return(segment_does_not_exist);
939          end;
940      if !sublock(temp_branch_l.bt_sbte.sbte_access, seg_access_p, ring_p, size(seg_access_p))
941          then return(invalid_argument_pointer)
942      else return(no_error)
943  end;

```

Actual Code of "get_seg_access"

Figure 9b

Implementation Code Written in UCLA Pascal for "get_seg_access"

```

944
945  function get_segment_info(      var      table_type_p      :seg_table_type;
946                                var      branch_p            :branch_table;
947                                mnt_inx_p          :mount_index;
948                                var      seguid_p           :seguid}      :branch_table;
949
950  var      register      aste_l      :aste;
951                                branch_l :      branch_number;
952
953  begin
954  #ifdef TFLAG
955  if (bit_test(tracing, TFUNCALL))
956  then trace("gtsginfo");
957  #endif
958  aste_l := find_aste(seguid_p);
959  if (aste_l != NULL)
960  then begin
961      if (bit_test(aste_l.aste_sbte.sbte_flags, SBTE_DELETED))
962      then return(NULL)
963      else begin
964          table_type_p := aste_type;
965          return(aste_l)
966      end
967  end;
968  branch_l := find_branch(mnt_inx_p, seguid_p);
969  if (branch_l = BRANCH_NOT_FOUND)
970  then return(NULL);
971  read_branch(branch_l, mnt_inx_p, branch_p);
972  if (branch_p.bt_sbte.sbte_seguid.uid_id[0] != seguid_p.uid_id[0])
973  then return(NULL)
974  else begin
975      table_type_p := sbte_type;
976      return(addr(branch_p))
977  end
978  end;

```

Code of Internal Function "get_segment_info" Used in "get_seg_access"

Figure 9c

The two visible exceptions of the VFUN, namely "invalid__segment__name" and "segment__does__not__exist," are mapped to the exceptions with the same name of the UCLA Pascal code found in statements "segment.p 908, 911, 922, and 938."

The only visible data structures are the parameters exchanged by the caller process and the VFUN module. The mapping of these parameters is shown in the header file at lines "def.h 370 and 372."

3.2 Unmapped Implementation Code

The following lines of UCLA Pascal code have no correspondent SPECIAL code:

- segment.p 888–898—Implementation language detail (i.e., declarations of function parameters and internal system data structures).
- segment.p 901–904 (and 954–957)—Conditional compilation of debugging code}.
- segment.p 905–906 and 940–942—Implementation code of the reference monitor mechanism (i.e., code that validates parameters passed by reference that helps maintain noncircumventability and isolation properties).
- segment.p 912–913, 920–921, 924–930—Implementation details referring to internal data structures that remain invisible to the user interface. Note the use of locking code, which ensures that internal sequences of kernel actions cannot be interrupted.
- segment.p 900,943—Implementation language detail (i.e., control statements).

3.3 List of Properties Preserved by the Mapping

The properties preserved by the mapping are:

- Mandatory Access Control to objects of type p segment.
- Equivalence of information flows visible at the TCB interface.

3.4 Justification for the Maintained Properties and for Unmapped Code

MAC Properties of Segments

In both the SPECIAL FTLS and UCLA Pascal code versions of "get_segment_access," control returns successfully to the invoking process only if the "valid_flow" and the "non_discretionary_access_allowed" checks pass. As explained above, these checks are equivalent from an unprivileged user's point of view. Furthermore, in both the FTLS and code versions, the unsuccessful returns are caused by the same sets of exception checks, namely (1) wrong object type, segment is not in the required file system partition (consistency checks); and (2) unmounted segment, failed MAC check, and inexistent segment (MAC relevant checks). The two additional exception checks present in the implementation code are not MAC specific checks. Instead, they are checks of the reference monitor mechanism (e.g., parameter validation), and thus irrelevant for MAC property verification.

Equivalence of Information Flows

The only visible flows of information through the interface of the "get_segment_access" module are those provided by the successful and the unsuccessful returns. These returns take place in identical FTLS and code conditions (namely, the mapping definition documented above and Figures 8; 9a; 9b, parts 1 and 2; and 9c). The additional exception returns of the implementation code to the invoker (i.e., the parameter validation exceptions) cannot introduce flows between different processes. Therefore, the equivalence of the FTLS (SPECIAL) flows and the implementation code (UCLA Pascal) flows is preserved.

Justification for Unmapped Code

The unmapped code cannot affect the mapping properties that must be preserved for the following reasons:

- The syntax of the parameter declarations and of the control statements are property irrelevant language details.
- The debugging code is not compiled in the TCB in the normal mode of operation.

- The code implementing the reference monitor checks is not specific to either of the above properties (although the functional correctness of these checks is required for secure system operation, such proof of correctness is not required for A1 systems currently).
- The code which implements internal kernel actions in a manner that cannot be interrupted is not visible at the TCB interface (although its functional correctness is required in secure systems, it is not always demonstrable using currently approved tools for A1 systems).

GLOSSARY

Access

A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

Administrative User

A user assigned to supervise all or a portion of an ADP system.

Audit

To conduct the independent review and examination of system records and activities.

Audit Trail

A set of records that collectively provides documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports and/or backwards from records and reports to their component source transactions.

Auditor

An authorized individual, or role, with administrative duties, which include selecting the events to be audited on the system, setting up the audit flags which enable the recording of those events, and analyzing the trail of audit events.

Authenticate

To establish the validity of a claimed identity.

Authenticated User

A user who has accessed an ADP system with a valid identifier and authentication combination.

Bandwidth

A characteristic of a communication channel that is the amount of information that can be passed through it in a given amount of time, usually expressed in bits per second.

Bell-LaPadula Model

A formal state transition model of computer security rules. In this formal model, the entities in a computer system are divided into abstract sets of subjects and objects. The notion of a secure state is defined and it is proven that each state transition preserves by moving from secure state to secure state, thus inductively proving that the system is secure. A system state is defined to be "secure" if the only permitted access modes of subjects to objects are in accordance with a specific security policy. In order to determine whether or not a specific access mode is allowed, the clearance of a subject is compared to the classification of the object and a determination is made as to whether the subject is authorized for the specific access mode. The clearance/classification scheme is expressed in terms of a lattice. (Also see Lattice).

Channel

An information transfer path within a system. May also refer to the mechanism by which the path is effected.

Covert Channel

A communication channel that allows a process to transfer information in a manner that violates the system's security policy. (Also see Covert Storage Channel and Covert Timing Channel.)

Covert Storage Channel

A covert channel that involves the direct or indirect writing of a storage location by one process and the direct or indirect reading of the storage location by another process. Covert storage channels typically involve a finite resource (e.g., sectors on a disk) that is shared by two subjects at different security levels.

Covert Timing Channel

A covert channel in which one process signals information to another by modulating its own use of system resources (e.g., CPU time) in such a way that this manipulation affects the real response time observed by the second process.

Coverage Analysis

Qualitative or quantitative assessment of the extent to which the test conditions and data show compliance with required properties, e.g., security model and TCB primitive properties, etc. (Also see Test Condition and Test Data.)

Data Integrity

The state that exists when computerized data are the same as those that are in the source documents and have not been exposed to accidental or malicious alteration or destruction.

Descriptive Top-Level Specification (DTLS)

A top level specification that is written in a natural language (e.g., English), an informal program design notation, or a combination of the two.

Discretionary Access Control (DAC)

A means of restricting access to objects based on the identity of subjects and/or groups to which they belong or on the possession of a ticket authorizing access to those objects. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) onto any other subject.

Dominate

Security level S1 is said to be the dominate security level if the hierarchical classification of S1 is greater than or equal to that of S2 and the nonhierarchical categories of S1 include all those of S2 as a subset.

Exploitable Channel

Any channel that is usable or detectable by subjects external to the Trusted Computing Base.

Flaw

An error of commission, omission, or oversight in a system that allows protection mechanisms to be bypassed.

Flaw Hypothesis Methodology

A system analysis and penetration technique where specifications and documentation for the system are analyzed and then flaws in the system are hypothesized. The list of hypothesized flaws is prioritized on the basis of the estimated probability that a flaw actually exists and, assuming a flaw does exist, on the ease of exploiting it and on the extent of control or compromise it would provide. The prioritized list is used to direct the actual testing of the system.

Formal Proof

A complete and convincing mathematical argument, presenting the full logical justification for each proof step and for the truth of a theorem or set of theorems. The formal verification process uses formal proofs to show the truth of certain properties of formal specification and for showing that computer programs satisfy their specifications.

Formal Security Policy Model

A mathematically precise statement of a security policy. To be adequately precise, such a model must represent the initial state of a system, the way in which the system progresses from one state to another, and a definition of a "secure" state of the system. To be acceptable as a basis for a TCB, the model must be supported by a formal proof that if the initial state of the system satisfies the definition of a "secure" state and if all assumptions required by the model hold, then all future states of the system will be secure. Some formal modeling techniques include state transition models, temporal logic models, denotational semantics models, and algebraic specification models.

Formal Top-Level Specification (FTLS)

A Top Level Specification that is written in a formal mathematical language to allow theorems showing the correspondence of the system specification to its formal requirements to be hypothesized and formally proven.

Formal Verification

The process of using formal proofs to demonstrate the consistency (design verification) between a formal specification of a system and a formal security policy model or (implementation verification) between the formal specification and its program implementation.

Functional Testing

The portion of security testing in which the advertised features of a system are tested for correct operation.

Lattice

A partially ordered set for which every pair of elements has a greatest lower bound and a least upper bound.

Least Privilege

This principle requires that each subject in a system be granted the most restrictive set of privileges (or lowest clearance) needed for the performance of authorized tasks. The application of this principle limits the damage that can result from accident, error, or unauthorized use.

Mandatory Access Control (MAC)

A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

Multilevel Device

A device that is used in a manner that permits it to simultaneously process data of two or more security levels without risk of compromise. To accomplish this, sensitivity labels are normally stored on the same physical medium and in the same form readable by machines or humans as the data being processed.

Object

A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, and network nodes, etc.

Process

A program in execution. It is completely characterized by a single current execution point (represented by the machine state) and address space.

Protection Critical Portions of the TCB

Those portions of the TCB, the normal function of which is to deal with the control of access between subjects and objects.

Read

A fundamental operation that results only in the flow of information from an object to a subject. Read Access (Privilege) Permission to read information.

Security Level

The combination of a hierarchical classification and a set of nonhierarchical categories that represents the sensitivity of information.

Security Policy

The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

Security Policy Model

An informal presentation of a formal security policy model.

Security Relevant Event

Any event that attempts to change the security state of the system, e.g., change discretionary access controls, change the security level of the subject, or change a user's password, etc. Also, any event that attempts to violate the security policy of the system, e.g., too many attempts to login, attempts to violate the mandatory access control limits of a device, or attempts to downgrade a file, etc.

Security Testing

A process used to determine that the security features of a system are implemented as designed and that they are adequate for a proposed application environment.

Single Level Device

A device that is used to process data of a single security level at any one time. Since the device need not be trusted to separate data of different security levels, sensitivity labels do not have to be stored with the data being processed.

Subject

An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair.

Subject Security Level

A subject's security level is equal to the security level of the objects to which it has both read and write access. A subject's security level must always be dominated by the clearance of the user the subject is associated with.

TCB-primitive

An operation implemented by the TCB whose interface specifications (i.e., names, parameters, effects, exceptions, access control checks, errors, and calling conventions) are provided by system reference manuals or DTLS/FTLS as required.

Test Condition

A statement defining a constraint that must be satisfied by the program under test.

Test Data

The set of specific objects and variables that must be used to demonstrate that a program produces a set of given outcomes.

Test Plan

A document or a section of a document which describes the test conditions, data, and coverage of a particular test or group of tests. (Also see Test Condition, Test Data, and Coverage Analysis.)

Test Procedure (Script)

A set of steps necessary to carry out one or a group of tests. These include steps for test environment initialization, test execution, and result analysis. The test procedures are carried out by test operators.

Test Program

A program which implements the test conditions when initialized with the test data and which collects the results produced by the program being tested. Top Level Specification (TLS) is a nonprocedural description of system behavior at

the most abstract level. Typically a functional specification that omits all implementation details.

Trusted Computer System

A system that employs sufficient hardware and software integrity measures to allow its use for simultaneously processing a range of sensitive or classified information.

Trusted Computing Base (TCB)

The totality of protection mechanisms within a computer system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy. It creates a basic protection environment and provides additional user services required for a trusted computer system. The ability of a trusted computing base to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.

Trusted Path

A mechanism by which a person at a terminal can communicate directly with the Trusted Computing Base. This mechanism can only be activated by the person or the Trusted Computing Base and cannot be imitated by those untrusted. Any person who interacts directly with a computer system.

Verification

The process of comparing two levels of system specification for proper correspondence (e.g., security policy model with top level specification, TLS with source code, or source code with object code). This process may or may not be automated.

Write

A fundamental operation that results only in the flow of information from a subject to an object.

Write Access (Privilege)

Permission to write to an object.

REFERENCES

1. Howden, W.E., "The Theory and Practice of Functional Testing," *IEEE Software*, September 1985, pp. 18-23.
2. Haley, C.J. and Mayer, F.L., "Issues on the Development of Security Related Functional Tests," Proceedings of the Eighth National Computer Security Conference, National Bureau of Standards, Gaithersburg, Maryland, September 1985.
3. Gligor, V.D., "An Analysis of the Hardware Verification of the Honeywell SCOMP," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, April 1985.
4. Gligor, V.D., "The Verification of the Protection Mechanisms of High Level Language Machines," *International Journal of Computer and Information Sciences*, Vol. 12, No. 4, August 1983, pp. 211-246.
5. Petschenik, N., "Practical Priorities in System Testing," *IEEE Software*, September 1985, pp. 1-23.
6. Myers, G.J., *The Art of Software Testing*, John Wiley and Sons, New York, 1979.
7. Howden, W.E., "Functional Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 162-169.
8. Clark, D., "Ancillary Reports: Kernel Design Project," M.I.T. Laboratory Computer Science, Cambridge, Massachusetts, Technical Memo 87, June 1977.
9. Gligor, V.D.; Chandrasekaran, C.; Jiang, W.D.; Johri, A.; Luckenbaugh, G.L. and Reich, L.E., "A New Security Testing Method and Its Application to the Secure Xenix Kernel," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 2, February 1987, pp. 169-183.
10. Laski, J.W. and Korel, B., "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983, pp. 347-354.

REFERENCES

11. Rapps, S. and Weyuker, E.J., "Data Flow Analysis Techniques for Test Data Selection," Proceedings of the Sixth International Conference on Software Engineering, 1982, pp. 272-278.
12. Luckenbaugh, G.L.; Gligor, V.D.; Dotterer, L.J.; Chandersekaran, C.S. and Vasudevan, N., "Interpretation of the Bell-LaPadula Model in Secure Xenix," Proceedings of the Ninth National Computer Security Conference, Gaithersburg, Maryland, September 1986.
13. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985 (supersedes CSC-STD-001-83, dtd 15 Aug 83), Library No. S225,711.
14. *Department of Defense—ADP Security Manual—Techniques and Procedures for Implementing, Deactivating, Testing, and Evaluating Secure Resource Sharing ADP Systems*, DoD 5200.28-M, revised June 1979.
15. Gligor, V.D.; Chandersekaran, C.S.; Jiang, W.D.; Johri, A.; Luckenbaugh, G.L. and Vasudevan, N., "Design and Implementation of Secure Xenix," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 2, February 1987, pp. 208-221.
16. Tsai, C.R. and Gligor, V.D., "A Bandwidth Computation Model for Covert Storage Channels and its Applications," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, April 1988.
17. Aerospace Report No. TOR-0086 (6777-25)1, "Trusted Computer System Evaluation Management Plan," 1 October 1985.
18. National Computer Security Center, *Trusted Product Evaluations—A Guide For Vendors*, NCSC-TG-002, Version-1, 22 June 1990.
19. Millen, J.K., "Kernel Isolation for the PDP-11/70," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, 1982.
20. Cugini, J.A.; Lo, S.P.; Hedit, M.S.; Tsai, C.R.; Gligor, V.D.; Auditham, R. and Wei, F.J., "Security Testing of AIX System Calls using Prolog," Proceedings of the Usenix Conference, Baltimore, Maryland, June 1989.

21. Millen, J.K., "Finite-State Noiseless Covert Channels," Proceedings of the Computer Security Foundation Workshop II, Franconia, New Hampshire, June 1989. (IEEE Catalog Number 89TH02550)
22. Carnall, J.J. and Wright, A. F., "Secure Communication Processor—Hardware Verification Report," Technical Report, Honeywell Inc., Program Code No. 7P10, prepared for Contract No. HAVELEX N00039-77-C-0245.
23. Honeywell, Inc., "Secure Communication Processor—Test and Verification Software Description," Technical Report, Rev. 3, April 1980, Program Code No. 7P10, prepared for Contract No. HAVELEX N00039-77-C-0245.
24. Vickers-Benzel, T., "Overview of the SCOMP Architecture and Security Mechanisms," The MITRE Corporation, Technical Report, MTR-9071, September 1983.
25. Abadi, M. and Lamport, L., "The Existence of Refinement Mappings," Research Report 29, Systems Research Center, Digital Equipment Corporation, August 1988.
26. Berry, D.M., "Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 2, February 1987, pp. 184-201.
27. Yu, C.F. and Gligor, V.D., "A Formal Specification and Verification Method for the Prevention of Denial of Service in ADATM Services," Institute for Defense Analyses, Paper No. P-2120, July 1988.
28. Vickers-Benzel, T., "Analysis of a Kernel Verification," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, April 1984.
29. Solomon, J., "Specification-to-Code Correlation," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, April 1982.

REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE July 1993	3. REPORT TYPE AND DATES COVERED Final	
4. TITLE AND SUBTITLE <i>A Guide to Understanding Security Testing and Test Documentation in Trusted Systems</i>		5. FUNDING NUMBERS	
6. AUTHOR(S)			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Security Agency Attention: I94; INFOSEC Standards, Criteria, and Guidelines Division 9800 Savage Road Fort George G. Meade, MD 20755-6000		8. PERFORMING ORGANIZATION REPORT NUMBER NCSC-TG-023	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER Library No. S-232,561	
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release: Distribution Unlimited		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The specific guidelines in <i>A Guide to Understanding Security Testing and Test Documentation in Trusted Systems</i> provide a set of good practices related to security testing and the development of test documentation. This technical guideline has been written to help the vendor and evaluator community understand what deliverables are required for test documentation, as well as the level of detail required of security testing at all classes in the <i>Trusted Computer System Evaluation Criteria</i> .			
14. SUBJECT TERMS coverage analysis, descriptive top-level specification, formal top-level specification, functional testing, security testing, test condition, test data, test plan, test procedure, verification		15. NUMBER OF PAGES 122	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT